Data Extraction HLD v0.2

Created by Neill Alexander, last modified on Nov 23, 2017

Purpose of the Document

This document outlines a high level design for providing data export and data access from EDH, or an EDH-like environment on AWS. It may be necessary to produce low-level designs for each of the components identified below, along with a UX design for the UI components.

Background

Export

The related CDS story is: http://10.102.81.252/browse/CDSP-1949

An export provides row level information about a declaration. No aggregation or analysis is done. The user can select the fields that he/she wants to include, some basic filtering (e.g. everything in February 2017), then run the export and download the results as csv.

User Initiated

For a user initiated export, a user goes to a web page, does something to trigger the export, and the results are downloaded synchronously as a CSV file.

Batch

Batch export is something that can be scheduled in advance, and will run without user interaction. In this scenario, we need a different delivery mechanism, since there will be no browser window sitting waiting to deliver the results. We can potentially support different delivery mechanisms e.g. UTM to CAF, email delivery (if not using EDH), etc.

Access

The related CDS story is: http://10.102.81.252/browse/CDSP-1925

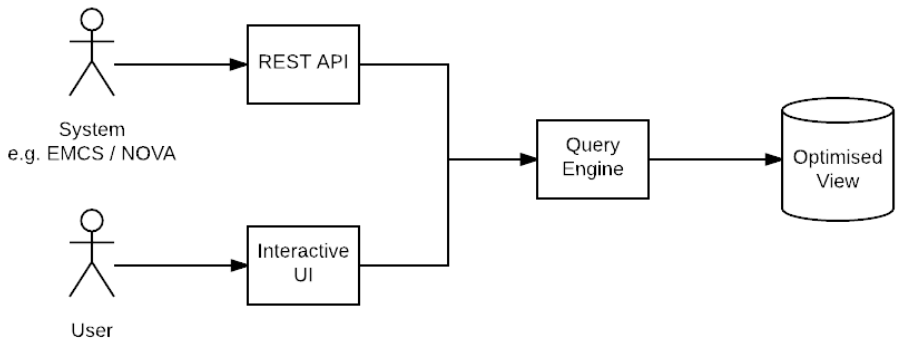
The specific requirements we have for this include:

1. NOVA Discovery Findings
2. EMCS Interface

For both of these, the existing solutions use the MSS Query Service to extract the data. We need to provide an equivalent service over the same data, callable by the NOVA and EMCS systems.

Moving to a higher level of abstraction, we need to be able to provide APIs which will return data from our Hadoop cluster, be that in EDH or AWS.

Data Extraction = Data Export + Data Access



Whilst doing the initial investigation into the Data Export and Data Access requirements, and considering the various approaches we could take, it quickly became clear that a large degree of overlap exists between the two requirements. They both entail running queries against some kind of optimised view of the data. In reality, one optimised view will support both Data Export and Data Access.

The main differences between Data Export and Data Access are:

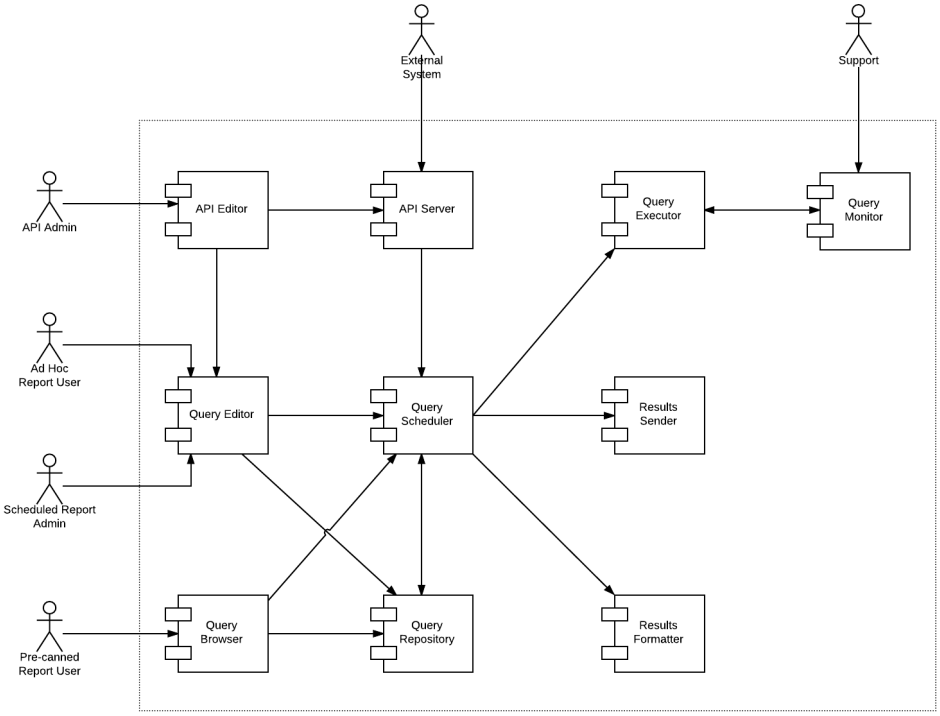
1. The method of initiating the query
2. The method of delivering the query results.

For simplicity, we propose to combine both sets of requirements into a single **Data Extraction** capability.

Potential Overlap With Search

In a separate stream of work we will also provide a search capability across declarations. Although there is clear overlap, in the sense that both will return lists of declarations meeting certain criteria, the intended use of each differs. Search provides a quick way to access declarations which meet certain criteria. It is not expected that every search result will be examined. Data extraction, by definition, returns **all** declarations which meet the specified criteria. A user would typically run this, then use the results for further analysis.

Component View



Query Editor

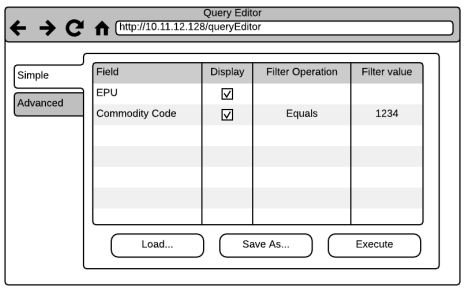
Responsibilities

- Provide a way for user to build and execute queries
- 2 views:
  - Simple
    - Allow users to build query visually
    - Select fields
    - Simple filters
    - Simple joins
  - Advanced
    - User can enter SQL
    - We should still make sure they don't try to do anything crazy
      - Perhaps parse the SQL and disallow certain potentially destructive constructs?
- User can choose to execute the query straight away
- Can also create / update / delete queries
  - Though, for delete, will need to take into account whether or not it is used in an API (see below)
- Can use the UI to set up a scheduled execution of the query
  - This would require allowing different delivery mechanisms, since wouldn't be interactive
    - Delivery to CAF directory
    - Email?

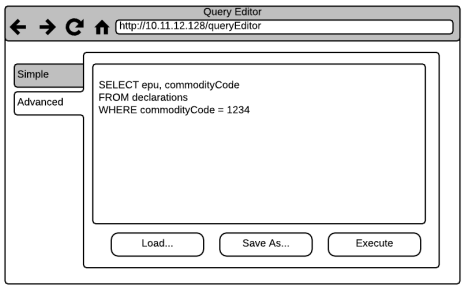
Visualization

NB: All UI visualizations are to communicate the general idea only. They are not UX designs to be slavishly adhered to.

Simple View



Advanced View



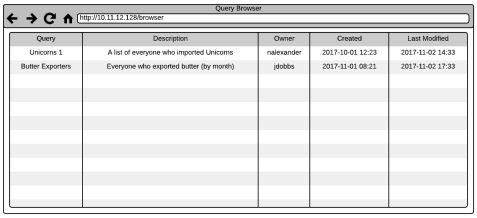
⚠ Whether or not an advanced view is warranted should be determined in collaboration with users. I would strongly advise including it, since it means that user will not be limited by potential shortcomings in the Simple view. For example, say they want to run a query with a filter type that we haven't provided in the Simple view. Without an Advanced view, they would need to wait for a new release of the software to be developed. With the Advanced view, they would have a workaround, even if it required help from a more advanced user.

Query Browser

Responsibilities

- Provides a view into the repository - can see all queries
- Can link to the editor from here
  - They might end up being the same physical component i.e. a web app to allow user to view / update etc queries

Visualization



For Further Investigation

A review of an earlier draft of this document brought up the following questions. These should be addressed in a low-level design document.

- Will it be possible to set permissions on pre-canned reports (e.g. for patricular person/people/department) in a way that is easy for a non-technical report creator? Will this be in the Query Editor?
- We need to think about how a user would find a report amongst hundreds/thousands. Needs UX thought. E.g. MSS Trade Reports ~1000 of the same report produced - one for each trader every month - would be ~1000 rows in the Query Scheduler just for that 1 report (or 1 with some code to insert the parameters for each trader?)

Query Repository

Responsibilities

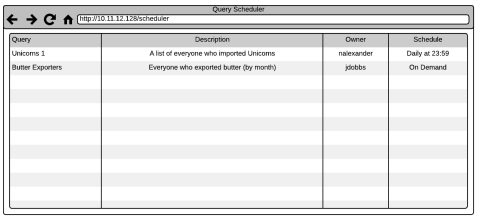
- This is where we store the queries
- Could be HBase or HBase + Phoenix
  - For online access needs to be quick
- Could be a RDBMS
- Provides a very simple CRUD interface

Query Scheduler

Responsibilities

- When to run a query
- Responsible for calling the executor at the correct time
  - Can be 'now'
- Will need it's own persistent store
  - Can we spin up a DB for this?
  - Or just use HBase + Phoenix?
- Will require a front end view and a way to CRUD schedules
- Called by the query editor to run an ad-hoc query on demand

Visualization



Query Executor

Responsibilities

- Query executor will in reality be a fairly simple component

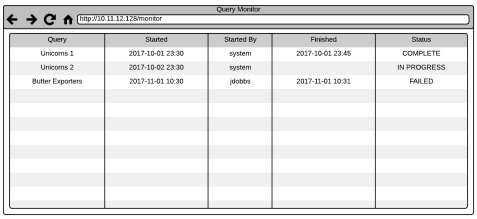
    - It will execute the SQL against Impala and return the result
- Assumption that Impala will be used to run these queries
    - It's designed specifically for this kind of thing
    - Gives ultimate flexibility in terms of what can be queried and how queries can be put together
    - Performance should not be an issue. Even 7 years worth of post-Brexit declarations is just about Big Data...
        - This will be validated by running some queries against production
        - And if over time it doesn't prove fast enough, we can provide further optimized layers for reports to run against, but keep the same architecture
            - Could also have custom executors for specific queries that need optimization
            - e.g. say there is an API published that is being called at very high volumes, we could provide a custom implementation as a Spark job, or something else, still referenced by name, and still with the same parameters.

Query Monitor

Responsibilities

- Keeps track of running queries
- Allows to see the history of queries that have been run
    - i.e. audit
- Also useful for support i.e. why did my query not run

Visualization



Results Formatter

Responsibilities

- Format results to desired output format
- Format can be specified dynamically
    - Start simple and add more formatters

Results Sender

Responsibilities

- Deliver the results to the user via there preferred method
- Import for the asynchronous exports

Results Repository

Responsibilities

- Store the results as they were sent out to the user
- Could be as simple as writing files to HDFS
    - Job to package them up on a weekly basis to avoid the 'small files' problem

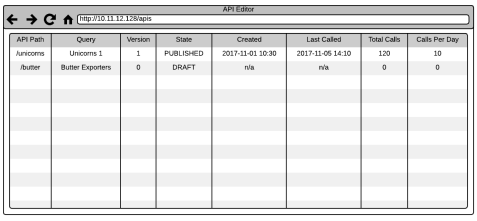
API Editor

Responsibilities

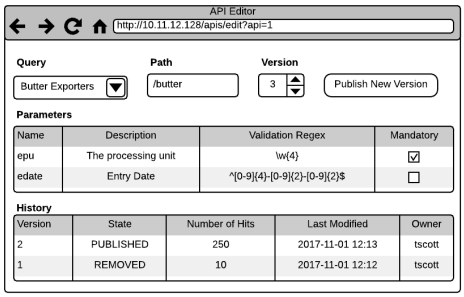
- Once we have a saved query, can use the API Editor to make it available via REST
- Would need checks to make sure that once an API has been published, the query can't be changed
    - Wouldn't want someone to accidentally change a query and break an API
    - Think the Query Editor should have that kind of control
        - i.e. a persisted query has metadata, including whether or not it has been published as an API
        - Once published, can't be revoked? Can only publish a new version?
            - /extract/[QUERY_ID]/[VERSION]?param1=blah&param2=hello
- Possible to unpublish an API
    - Can provide stats on how often it has been called etc to check if it is still being used
- Define the permission required to connect via the API here

Visualization

List APIs



Edit APIs



NB: in the screenshot I included 'Validation Regex'. We could provide a simple view of this also, where we define parameter types, and then validate accordingly. Only power users would manipulate regular expressions directly.

API Server

Responsibilities

- Make a pre-defined query callable via a REST api
- Query is given a unique name, and that's what is used to execute it
- Can also be given a version to protect against changes
- Format the response to the required output

Example

Say I have defined a query like this:

```
select
a.iekey
,c.cmdtycode
,a.dtofent
,b.ecsu1
,b.ecsu2
,a.epsid
,a.epuno
,b.ieitno
,a.impentno
,a.imptrturn
,b.itemcstmsval
,b.itemnetmass
,b.itemsuppunits
,b.itemthrdqty
,c.origcntry
from imeidetail b,imeiselect c, imenselect a
where b.ieitno = c.ieitno
and b.iekey = c.iekey
and a.iekey = c.iekey
and c.cmdtycode = [COMMODITY]
```

This has been saved as a query called "commodities". Note that it expects a single parameter.

An external system would call this with something like:

- GET http://[API_SERVER_HOST/query/commodities/v1?COMMODITY=1234&format=csv

The API server would:

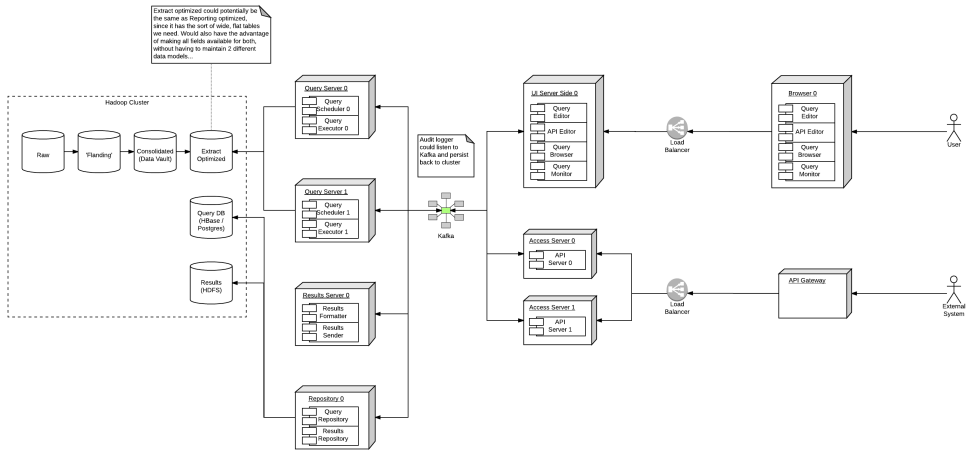
- Look up the query called 'commodities'
- Substitute the parameters
  - Some validation required here to protect against SQL Injection attacks
- Execute the query
- Output as the requested format

API Gateway

Would sit in front of the API server and do the access rights checking etc.

Deployment View

Here's an example of how the components could sit together with our existing infrastructure. I grouped Query Scheduler and Query Executor together because at a high level it seemed that they could be implemented in the same service. However, they could just as easily be kept completely separate. The various UI Server components equally could either be deployed together, or deployed as separate services.



Kafka

I've suggested using Apache Kafka as the communication layer between components. This would allow us to horizontally scale any aspect of the system by simply adding more consumer to a particular consumer group.

- Say we have 2 consumers in a 'query_executor' consumer group
- All messages will be distributed across the 2 consumers.
- If we require more capacity then we can add another consumer, and Kafka will now distribute messages across the 3 consumers.

For request/response type behaviour, we could have a 'query' topic and a 'query_response' topic, where some kind of structured document containing a correlation id is sent as the message to the 'query' topic, and the sender then listens on the 'query_response' topic and watches out for the matching response.

Example Query

```
{
    "correlationId": "query_123",
    "querySql": "SELECT ....",
    "userId": "jsmith"
}
```

Example Response

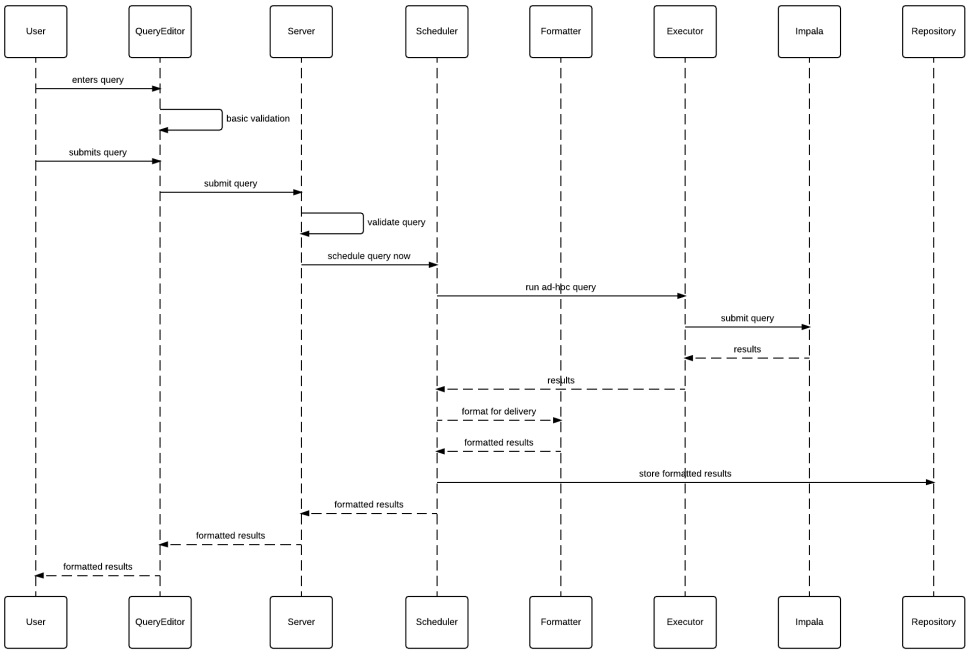
```
{
    "correlationId": "query_123",
    "response": [
        ["2017-01-01", "ID1", "User 1", "Notes"],
        ["2017-01-01", "ID2", "User 2", "Notes"]
    ]
}
```

For this to work, all requesting components would need to listen to all responses, and ignore those they aren't interested in (i.e. no consumer groups for the 'query_response' topic).

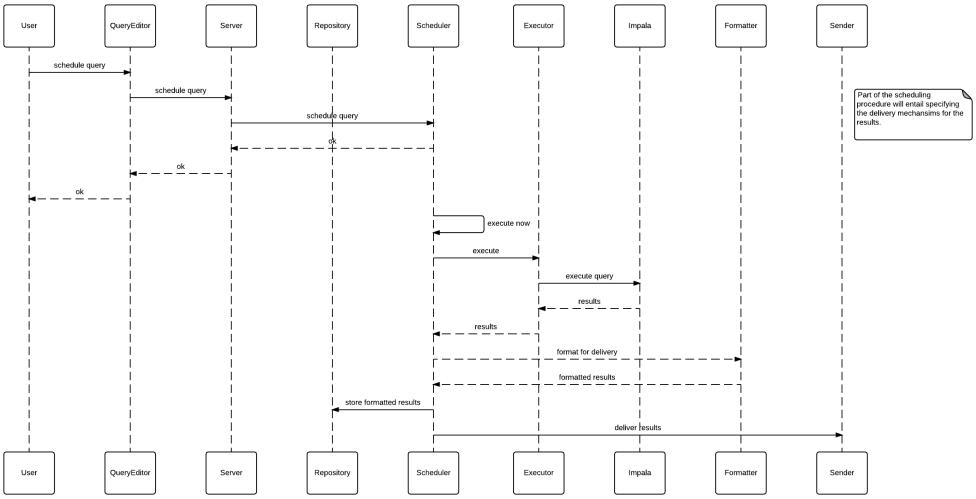
Sequence View

Following are some sequence diagrams to illustrate how the various components could interact to support the requirements.

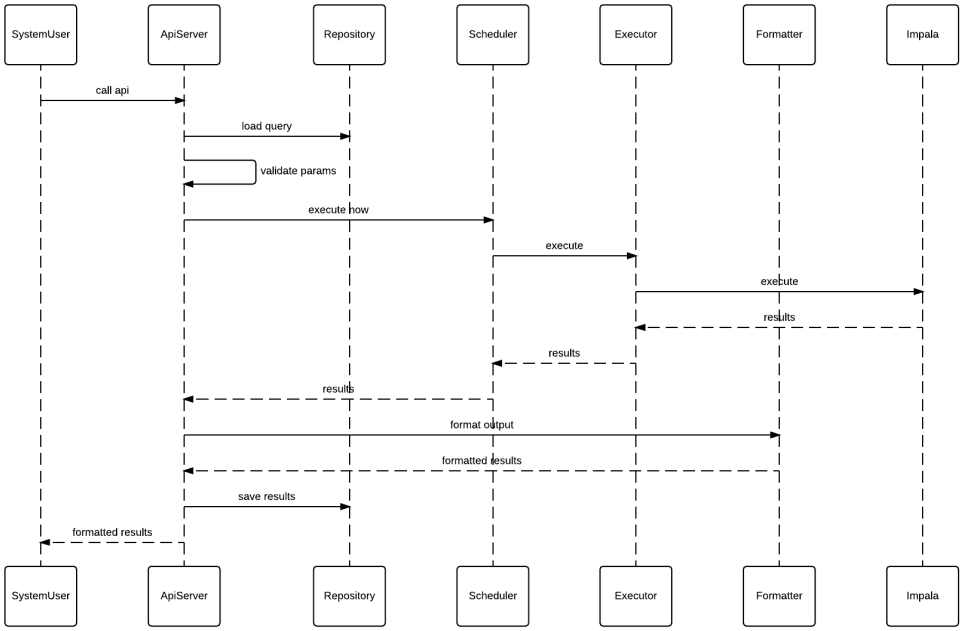
User Initiated Export



Scheduled Export



Data Access



Delivery Mechanisms

These will matter particularly with scheduled data export. For user-initiated export the hope would be that the query runs quickly enough that the user can simply wait for it to finish and then save a file to their local machine, via browser download. However, since we will need to support different delivery methods for the scheduled exports, we could also support them for user-initiated exports.

In all cases, we should store all data export files for a period of time in HDFS (it is a big data system after all, so let's store all the data!).

CAF

Copy file to a shared folder. We could use UTM to achieve this. The user would then know to expect the file in a specific location using a specific file format.

Email

Since the data export is internal only (for now) we could probably use email to send files. Or, if we can't send the files, send a link which would allow a user to download the file, providing they have the correct authorisation to access it.

Download File

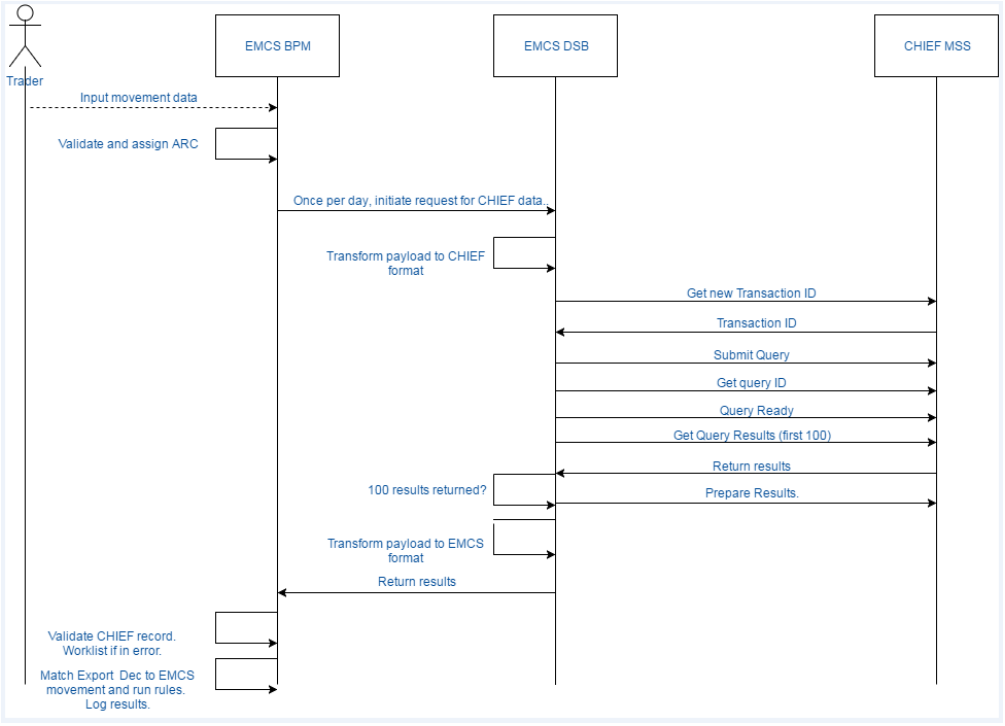
For user-initiated export, the browser could block and download the file once the results have been returned. We could also provide a link via the Query Monitor page which would allow the user to download the results - clicking on the link would require us to retrieve the file from HDFS and return to the user.

In terms of the data access API, we could provide an interface similar to the UN Comtrade Bulk Data API, which returns a response which provides a link to a file to download:


```
[
   {
      "name":"type-C_r-all_ps-2013_freq-A_px-S1_pub-20150318_fmt-csv_ex-20150318.zip",
      "r":"ALL",
      "px":"S1",
      "ps":"2013",
      "type":"COMMODITIES",
      "freq":"ANNUAL",
      "publicationDate":"2015-03-18T00:00:00",
      "extractDate":"2015-03-18T00:00:00",
      "filesize":179461899,
      "downloadUri":"http://comtrade.un.org/api/get/bulk/C/A/2013/ALL/S1"
   },
   {
      "name":"type-C_r-all_ps-2013_freq-A_px-H2_pub-20150314_fmt-csv_ex-20150314.zip",
      "r":"ALL",
      "px":"H2",
      "ps":"2013",
      "type":"COMMODITIES",
      "freq":"ANNUAL",
      "publicationDate":"2015-03-14T00:00:00",
      "extractDate":"2015-03-14T00:00:00",
      "filesize":372954375,
      "downloadUri":"http://comtrade.un.org/api/get/bulk/C/A/2013/ALL/H2"
   },
   ...
]
```


HTTP

For the data access API, we could return a synchronous or asynchronous response. If synchronous, the call simply blocks until results are available. If asynchronous, we would need to return a token to the calling system which would allow it to get the results at a later date. This kind of mechanism is used currently for the CHIEF MSS query API, with the transaction ID:



(reproduced from EMCS Interface)

Delivery Formats

Applicable to both data access (via the API) and export (via the UI / scheduler).

JSON

Something like:

```
{
    "queryId": "1234_xyz",
    "fields": {
        "fieldA": "fieldA_value",
        "fieldB": "fieldB_value"
    }
}
```

XML

Something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<queryResults>
   <queryId>1234_xyz</queryId>
   <fields>
      <field name="fieldA">fieldA_value</field>
      <field name="fieldB">fieldB_value</field>
   </fields>
</queryResults>
```

CSV

Something like:

```
fieldA, fieldB
fieldA_value, fieldB_value
```

MSS Legacy (maybe)

If existing users of the CHIEF MSS Query interface don't have the capacity to change the way they process the returned data, then we may need to replicate the output format, which is described at EMCS Interface as: "Data is returned in a MSS Enterprise specific format. (EMCS DSB transforms the data from the format output by MSS to a format that is used by EMCS BPM.)"

Ideally we would not need to do this.

Security Considerations

- Use Kerberos / RBAC to control access to declarations.
- Could introduce an API Gateway to control access to the API Server

No labels