Welcome

Thank you for taking an interest in the Open Client Registry! This is a community project and meant for others to adopt to their use cases as they wish.

To the left you'll see the Table of Contents for the documentation. On the right are the sections of each page.

The documentation is divided into the **User Manual** and **Developer Manual**.

How to use this site effectively

Please read the User Manual first as it introduces the basics. Here are some suggestions about how to use this documentation depending on your role or interest.

- **Implementer**: The workflow and decision rules pages explain the record linkage process and have links to resources to help teach others how to understand it and run it in an organization.
- **Data manager or analyst** The decision rules page is an overview of how matching is configured and discusses how algorithms are selected and chained together.
- **Developer** or **systems administrator**: The installation and configuration pages go into more detail about setup and configuration.

> **Tip**
>
> Regardless if you're just curious, an implementer, or a developer, please read the OpenCR Overview tab first and then the User Manual. We've kept them short.

Last update: February 20, 2020

What is OpenCR?

OpenCR is an open source and standards-based client registry. A client registry facilitates the exchange of patient information between disparate systems. A client registry holds patient identifers and may include patient demographic information. It is a necessary tool for public health to help manage patients, monitor outcomes, and conduct case-based surveillance.

A client registry sits within a health information exchange (HIE). An HIE is used to safely and effectively exchange information. A critical component of an HIE are registries, such as those to manage a shared, canonical facility list, practitioners, and patients.

What does OpenCR do?

OpenCR is offers the ability to:

- Assign and look-up unique identifiers,
- Allow connections from diverse point of service (POS) systems, such as electronic medical record (EMR) systems, that can submit messages in FHIR,
- Configure decision rules around patient matching.

> **Caution**
>
> This implementation does not allow point-of-service systems to get patient demographic information stored in the Client Registry. This is also not a Shared Health Record, nor does it contain patient clinical data.

Use Cases

The Client Registry is one component in a more complex HIS architecture needed to accomplish important use cases, such as:

- **Deduplicating patients**: Sometimes patients have multiple diagnostic results stored within a POS. The Client Registry will link patients based on configurable decision rules so multiple test results for the same patient can be found.
- **Tracking patients lost to clinical care**: EMRs are often not interoperable with one another, resulting in difficulty tracking patients as they move between facilities to seek care. A Client Registry will help data managers to track patients, decreasing instances of duplicate and incomplete records, patients LTFU, and sub-optimal care.

> **Caution**
>
> The Client Registry is not deduplicating or even touching patient clinical and demographic records within point-of-service systems. Instead, it provides a way to enable use cases like deduplication - which must be an external process.

Last update: February 20, 2020

Matching process

Insert block diagram

Last update: February 20, 2020

Use Cases

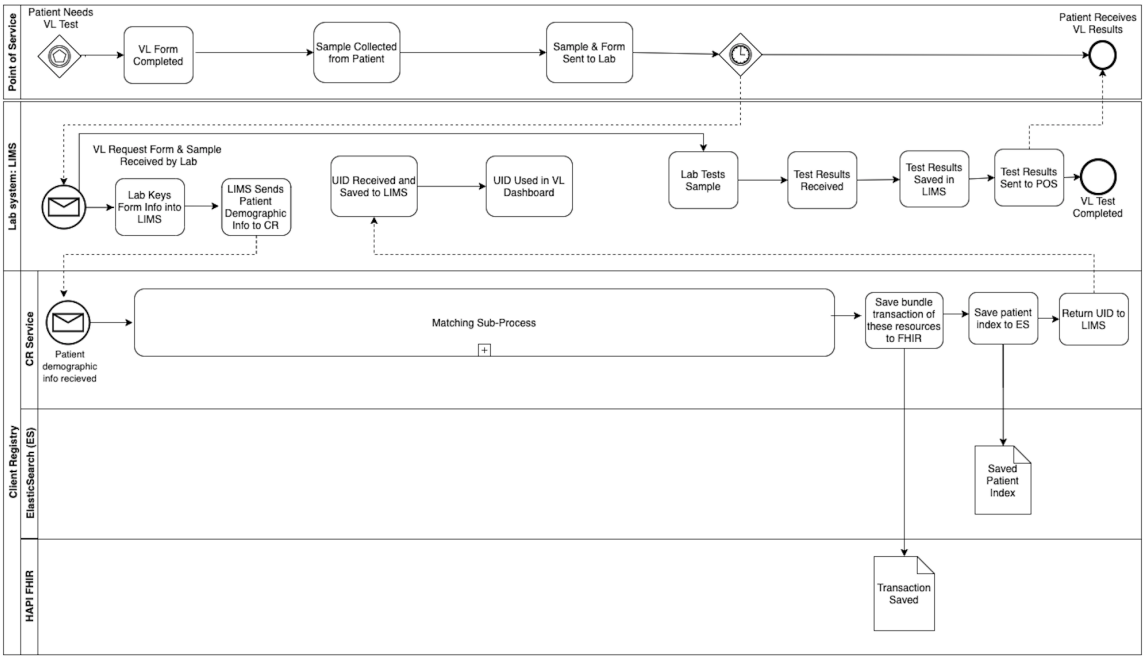
At its core, the Client Registry provides a unique identifier (UID) that also links to all other already matched records from submitting systems. This means that the Client Registry stores an identifier from submitting systems so that it can uniquely identify according to however the submitting systems store their records, but it also produces a UID for the entire domain using the service.

Several workflows are supported out-of-the-box depending on the POS-Client Registry use case. For example:

- A specimen is received by a laboratory. Demographic data and requesting location data is entered into the LMIS. The LMIS queries the Client Registry for a UID. The Client Registry provides the UID if one did not exist and stores limited patient demographic information but **does not** store test results. A use that this enables (but the Client Registry **does not** provide) is the ability to track persons lab results over time.
- A patient is registered at a clinic. The clinician recommends a viral load test. The specimen is sent for processing to the laboratory. The Client Registry receives the UID and specimen and returns a diagnostic result that is then stored in the EMR.
- A patient is registered at a clinic and has been assigned a UID. In the course of their clinical encounter, a sentinel event occurs, triggering the EMR to send limited clinical information to the Health Information Exchange (HIE). The HIE sends the data to a data analysis warehouse for population analysis and case-based surveillance.

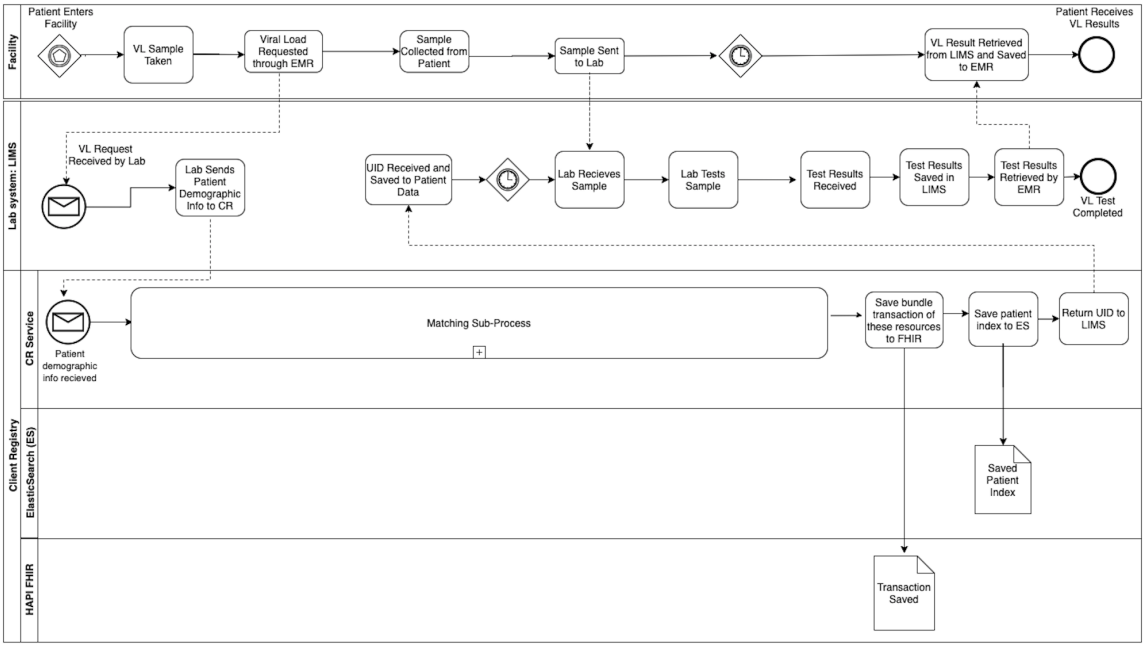
> **Warning**
>
> It is important to note that in the above workflows the Client Registry does not store or provide clinical data. Such processes are external to the Client Registry and must be separately created, governed, and enabled.

Viral Load Test Request (Paper)



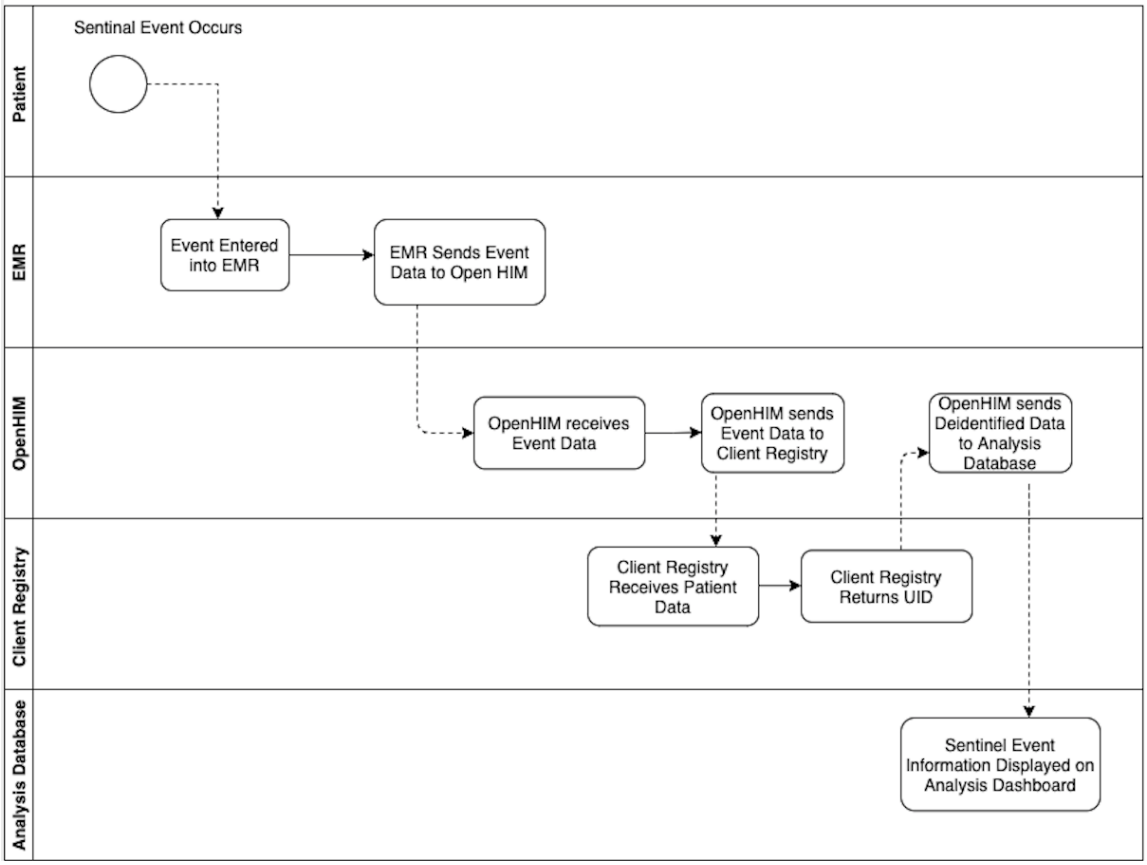
A plasma specimen is received by a laboratory for HIV viral load testing. Demographic data and requesting location data is entered into the LMIS. The LMIS queries the Client Registry for a UID. The Client Registry provides the UID if one did not exist and stores limited patient demographic information but **does not** store test results. A use that this enables (but the Client Registry **does not** provide) is the ability to track persons lab results over time.

Viral Load Test Request (EMR)



A patient is registered at a clinic. The clinician recommends an HIV viral load test. The plasma specimen is sent for processing to the laboratory. The Client Registry receives the UID and specimen and returns a diagnostic result that is then stored in the EMR.

Case-Based Surveillance



A patient is registered at a clinic and has been assigned a UID. In the course of their clinical encounter, a sentinel event occurs, triggering the EMR to send limited clinical information to the Health Information Exchange (HIE). The HIE sends the data to a data analysis warehouse for population analysis and case-based surveillance.

Last update: February 20, 2020

Intro to Algorithms

Algorithms can be deterministic, fuzzy, probabilistic, or a combination of approaches; most include some element of automatic matching plus human review. No perfect algorithm currently exists. Typically, the creator(s) of the client registry will agree on the appropriate threshold to determine a “match” and this may need to be recalibrated over time.

Approaches

Types of matching approaches:

- Deterministic. This can be thought of enforcing exact matches. This type of matching performs poorly when the quality of the data is low and/or the discriminatory power of the variable used for matching is low (e.g. sex is a poor discriminatory variable). If exact matching is enforced, the data scientist should verify that other fields also match using a combination of approaches, as needed.
- Fuzzy. The fuzzy match method relies on converting items to their core components (e.g. phonetics) to determine matching rather than relying on probabilities. This method’s strength is addressing typing errors but remains imperfect in cases where words are similar, but are not the same (e.g. there, their). Its efficacy is unproven in most languages other than English.
- Probabilistic. This is a method that assigns a match probability to a pair of records. The most common method (Fellegi-Sunter) checks a number of fields and sums up the probabilities that each field is a match. Higher scores indicate a match. Recent research has been directed at improving the FS method.

Matching algorithms are evaluated based on their sensitivity and specificity in detecting matches. One caveat to remember is to check whether the sensitivity and specificity reported are with or without human review. Generally, it is better to minimize the chances of a false positive; however, as false positives go towards zero false negatives increase and the balance must be maintained.

Types of Deterministic Matching

Field-based matching. Typically fields are compared to each other across data sets and a result is a match or non-match, thus deterministic.

Types of Fuzzy Matching

Methods that assist in matching based on phonetic transformations to remedy spelling errors: New York State Identification and Intelligence System (NYIIS) is a phonetic encoding of names that can assist with matching. It can only be used for English names. Soundex indexes names by sound as pronounced in English. The Soundex encoding includes the first letter of a surname followed by 3 digits. This method relies on correctly recording the first letter of a name. Metaphone indexes words by their English pronunciation. Double Metaphone implements Metaphone but allows for two encodings to be returned for a word rather than constraining it to one choice. It works well with names of various origins, but the lettering must be English. Metaphone 3 is the next generation of Double Metaphone and is a commercial product.

Methods that compare string similarity: Levenshtein edit distance is a measure of the number of edits (insertions and deletions) that would be required to change one string to another. Jaro-Winkler comparator/distance is similar to Levenshtein, but it gives more weight to strings that match on the beginning of the string. 0 is an exact match. Lower scores are better than higher scores. If the beginning of the string is misspelled, this will not work as well. Longest common subsequence matches subsequences within strings. Subsequences do not need to match positions.

Types of Probabilistic Matching

The most widely used probabilistic matching method is the Fellegi-Sunter method. In this method, each field is weighted by its discriminatory power and data quality; could be manually or via model training. The algorithm checks each field and decides whether the field matches between the pairs or not. If the fields match, the weight is added to the final score. If the fields do not match, the

weight is subtracted from the final score. If the sum of the weights is above a threshold, they are considered a match. Possible matches can also be produced and these would require human review.

One drawback is that this method assumes that no fields are correlated with each other or that they are not conditionally dependent. The assumption of conditional independence does not always hold and can lead to suboptimal results which is the biggest limitation of this method. For example, there are more women named Barbara than men named Barbara. First name and sex are not independent.

Extensions to the F-S method include: The approximate comparator extension (ACE) which includes string matching strategies prior to performing the weighting and scoring.

Alternatives to F-S include (see video): GHC Scaling Algorithm (Goldstein, et al. 2017, Stats in Medicine) an order of magnitude faster than F-S. Calculates underlying weights and doesn't rely on independence assumption.

Speed

Methods to reduce the number of comparisons and speed up the process: Blocking. This is like sorting socks by color before trying to match them. It speeds up the process by reducing the number of pairs that need to be checked.

Last update: February 20, 2020

Supported Algorithms

[Add table of algos and definitions] [link to algo background]

Mediator 1.Metaphone 2.Levenshtein 3.Exact

Standalone 1.levenshtein 2.damerau-levenshtein 3.jaro-winkler 4.soundex 5.Exact 6.metaphone 7.double-metaphone

---

Last update: February 20, 2020

Architecture

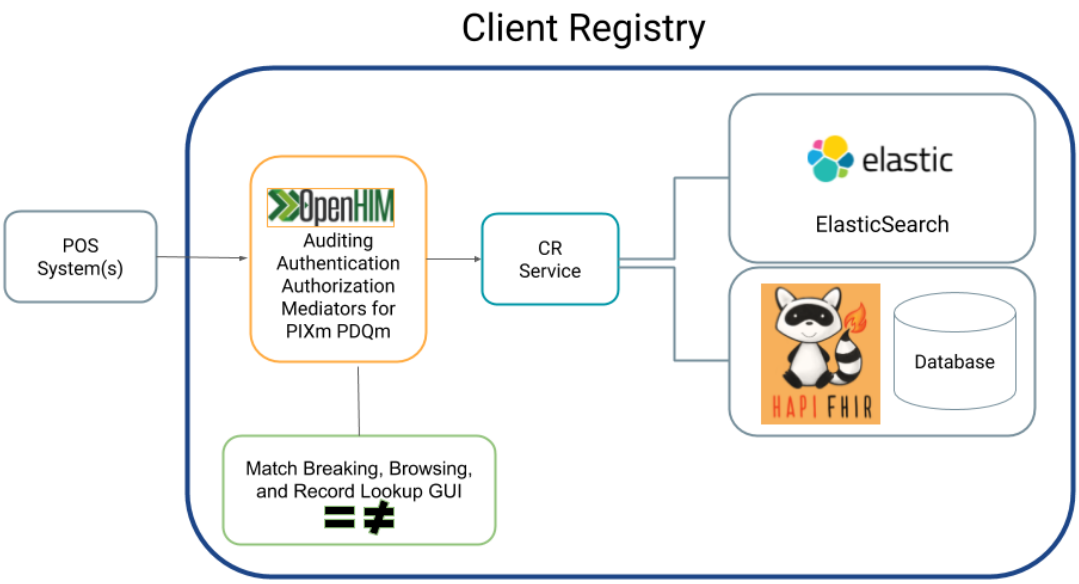
The Client Registry is not one application, instead it's a set of applications that work together in the Open Health Information Exchange (OpenHIE) architecture to serve point-of-service systems, like EMRs, insurance mechanisms, and labs.

> **Caution**
>
> This is not an OpenHIE product. It is a prototypical client registry to facilitate discussion among a broad set of stakeholders.

The architecture is made up of both a lite and a production version. In the production version it includes:

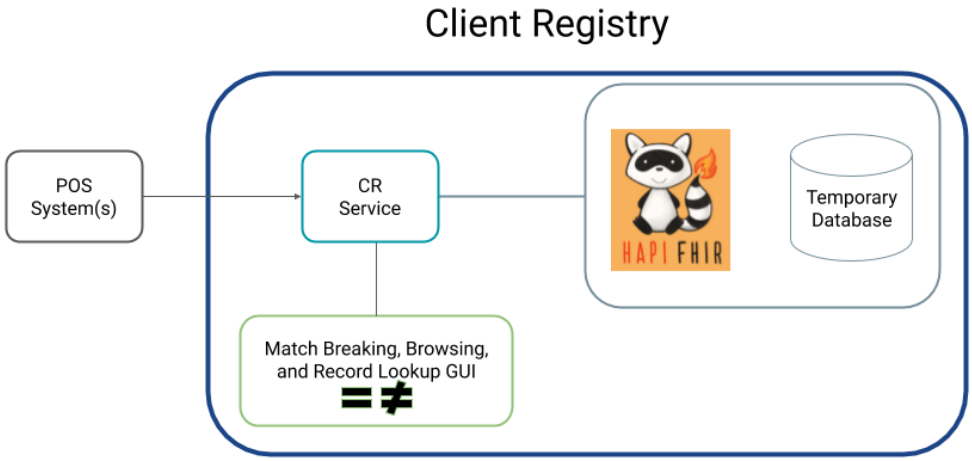
- The **Open Health Information Mediator (OpenHIM)**: The OpenHIM is the entrypoint for POS systems, and includes authentication (are you who you say you are?), authorization (what roles do you have permission to fulfill?), and auditing of all transactions.
- The **HAPI FHIR Server**: HAPI is the reference FHIR server in Java and scalable into production environments.
- The **ElasticSearch**: Elasticsearch is a powerful search engine that is highly performant.
- An **optional UI** to view and break matches between records, and to select and chain together decision rules around matching algorithms.

Client Registry Production Architecture



The lite architecture is for non-production and does not require OpenHIM or ElasticSearch.

Client Registry Lite Architecture



Last update: February 20, 2020

Introduction

OpenCR is an open source and standards-based client registry. A client registry facilitates the exchange of patient information between disparate systems. A client registry holds patient identifers and may include patient demographic information. It is a necessary tool for public health to help manage patients, monitor outcomes, and conduct case-based surveillance.

A client registry sits within a health information exchange (HIE). An HIE is used to safely and effectively exchange information. A critical component of an HIE are registries, such as those to manage a shared, canonical facility list, practitioners, and patients.

What does OpenCR do?

OpenCR is offers the ability to:

- Assign and look-up unique identifiers,
- Allow connections from diverse point of service (POS) systems, such as electronic medical record (EMR) systems, that can submit messages in FHIR, and
- Configure decision rules around patient matching.

> **Caution**
>
> This implementation does not allow point-of-service systems to get patient demographic information stored in the Client Registry. This is also not a Shared Health Record, nor does it contain patient clinical data.

The process for a point-of-service system like an EMR to get a unique ID from the Client Registry is straightforward though it looks complicated at first.

1. A POS provides some demographic information to the Client Registry.
2. The Client Registry looks for an existing record matching that patient.

3. If there is an existing record, the Client Registry provides the unique ID back to the POS.
4. If there is not an existing record, the Client Registry makes a new one and provides a unique ID back to the POS.

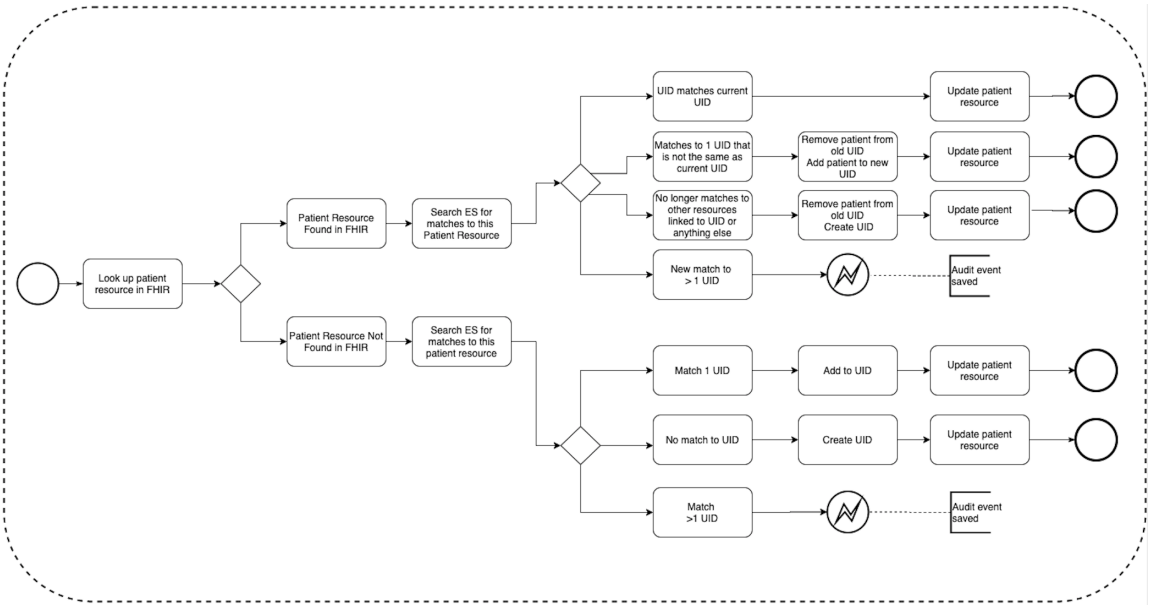
As noted in the introduction, the Client Registry provides a unique identifier that also links to all other already matched records from submitting systems. This means that the Client Registry stores an identifier from submitting systems so that it can uniquely identify according to however the submitting systems store their records, but it also produces a UID for the entire domain using the service.

> **Warning**
>
> The below workflows the Client Registry does not store or provide clinical data. Such processes are external to the Client Registry and must be separately created, governed, and enabled.

Last update: February 20, 2020

Record Linkage Process

The below diagram shows how OpenCR performs record linkage after the matching process. The diagram begins with a source system submitting a request with patient demographic data in a FHIR message, as indicated by the circle on the left.



After requests are submitted with demographic data, OpenCR reads the submitting system's ID of that patient. The Client Registry searches for that source system's ID in its records. This happens regardless if it is a new patient or update of existing patient.

When the submitting system's ID matches an existing record, the Client Registry updates the patient demographic information of that record with changes submitted. Once the update is complete, the existing record linkages may affected. This is because algorithms may not continue to link records as before because details have changed. Therefore, the Client Registry will pool all patients that were previously matched and break all the matches. The Client Registry will rerun matching algorithms again to see what matches are currently true matches of the patient. Then the Client Registry will be updated with the true matches given the changes in demographic data.

Another scenario is when the Client Registry searches and doesn't find anyone already with same submitting system's ID. If there is not existing match, the Client Registry runs the matching algorithms for existing patients who matches that patient and will provide record linkages with other records.

Requirements

In order for this process to work as expected, there are some requirements:

- Requests sent to the Client Registry must be made of FHIR messages. FHIR is a popular specification for accessing an API for providing data in health systems. Messages must support FHIR R4.
- Requests can only be received from trusted systems. See the security page in the Developers Manual for mode detail.

Last update: February 20, 2020

Decision Rules

Overview

Demographic data from submitting systems is stored in HAPI FHIR. It is also recommended that the demographic data that is primarily stored in HAPI FHIR be indexed into Elasticsearch.

For match processing, there are two options. One is run in mediator-only mode, which is highly flexible and supports a handful of algorithms that can be chained together. Additional algorithms can be added as needed.

The second is to use ES. ES is very fast and supports compound queries but currently only supports Levenshtein distance. When using ES, every request to the FHIR Server is cached in ES.

(One additional caveat for Levenshtein distance is that the mediator-only matching can support edit distances exceeding two, while ES edit distance cannot exceed two.)

Every client wishing to use the Client Registry must be authenticated and authorized. See the configuration page for more information.

How to Set Decision Rules

Decision rules determine how matches are made among records, for example, by using a certain algorithm on one field and a different algorithm on another.

Let's use the below example:

`rules.givenName` is used as one rule on the field givenName.

`rules.givenName.algorithm` defines an algorithm, in this instance Jaro-Winkler, and an threshold for that algorithm unique to it.

`rules.givenName.path` is a required FHIRpath for the fields, a standard way to define how to traverse a FHIR resource. In future, a GUI may be used for defining the FHIRpath.

By default, all of the rules are chained together in a logical AND statement. In ES the search queries are assembled into compound queries.

Link to file

Contents of `server/config/decision_rules.json`

```
{
  "__comments": {
    "path": "Its a fhir path, for syntax refer to https://
www.hl7.org/fhir/fhirpath.html",
    "type": "String, Date, Number or Boolean",
    "threshold": {
      "levenshtein": "Lower the number, the closer the match, 0
being exact match",
      "jaro-winkler": "number between 0 and 1, where 0 for no match
and 1 for exact match"
    }
  },
  "rules": {
    "givenName": {
      "algorithm": "jaro-winkler",
      "threshold": 0.89,
      "path": "name.where(use='official').last().given",
      "type": "string",
      "systems": ["system1", "system2", "system3"]
    },
    "familyName": {
      "algorithm": "damerau-levenshtein",
      "threshold": 3,
      "path": "name.where(use='official').last().family",
      "type": "String"
    },
    "gender": {
      "algorithm": "exact",
      "path": "gender",
      "type": "String"
    }
  }
}
```

Last update: February 16, 2020

Matching Benchmarks

Last update: February 16, 2020

Admin Interface

Last update: February 16, 2020

Dashboard

Kibana Dashboard

Under construction.

Last update: February 16, 2020

OpenHIE Implementation Guide

Implementation processes are complex. The OpenHIE Client Registry Subcommunity is a valuable resource for understanding the policies and practices of client registries, particularly in low resource settings. Visit their site.

The community provides a comprehensive resource, The Client Registry Implementation Guide, may be consulted for the overall processes required to implement, including how to:

- Analyze the current environment
- Establish leadership and governance
- Document specifications and requirements
- Implement specifications
- Create a support plan
- Conduct a post-production evaluation

Last update: February 17, 2020

Roles and Responsibilities

Responsibilities

There are a great deal of responsibilities that must be addresses for a successful implementation beyond the OpenHIE Implementation Guide.

- **Which systems will connect to the CR and how will support for querying the CR be implemented on the POS side?** There is emerging FHIR support for POS systems but features will need to be added to submit and process the queries. The Developer Guide includes a link to a reference implementation for OpenMRS MPI Client.
- **Which form fields of demographic data will be submitted?** Every use case and every form is different. There may be many different sets of demographic data that are stored, and this affects how matching is done.
- **Which algorithms and decision rules make sense for the use case?** There is a scientific literature on which algorithms perform efficiently for matching. There is a need to test algorithms – which can be done outside of the CR such as in R and Python – and the need to evaluate against what the CR implementation is doing to ensure consistency.
- **How will the matching algorithms be implemented, deployed, and backed-out for incorrect matches?** The CR includes an Admin UI for matching. The UI is meant to be highly restricted; it makes available demographic records.
- **Who is responsible for providing preprocessed data to the CR?** The CR accepts formatted FHIR messages. It does not impose its own algorithms for cleaning. Connecting POS systems must provide data in the correct format and preprocess the data before sending. Updates can be made to incorrect demographic data later and those will be added to existing records.
- **How will systems launch and scaling up be managed?** What network and compute resources are available for deployment. Advanced Linux systems administration skills are required to launch and maintain the CR in production.

Roles

In actual practice, there are specific roles.

- **Point-of-service systems users**: POS systems use the Client Registry to obtain a CRUID. This process is mostly invisible. Users of EMRs and other systems submitting queries may only see that there is a CRUID for a patient. The Client Registry is invisible to them.
- **POS developers**: Client Registry integration must be added into POS systems for them to be able to query for a CRUID. Software developers of POS systems should review the Developer Manual and understand how to implement the proper FHIR query for obtaining a CRUID.
- **Matching administrators** There may be situations in which the Client Registry implementation uses the UI for viewing and breaking matches. This is a privileged role that should be restricted to few individuals.
- **Client Registry systems administrator**: People managing the network, servers, backups and other aspects of the Client Registry. They should be very familiar with the Developer Guide, particularly security of the system, how to perform upgrades, and recovery procedures
- **Management team**: Governance of the system should be handled by a management team familiar with the implications of decisions, strategy, roll-out, and other aspects.

Last update: February 17, 2020

Additional Resources

Last update: February 16, 2020

Introduction

Last update: February 20, 2020

Internals

ElasticSearch

ES is a web service around the Apache Software Foundation-supported Lucene search engine. ES provides a JSON-based REST API, cluster management, and other value-add on top of Lucene. Many of the features discussed below are actually in Lucene, and are not specific to ES, although these docs refer to ES as including Lucene features. This can be confusing. ES means ES which includes Lucene.

Mapping

Data fields to be indexed in ES require that they first be mapped. The CR mediator takes a mapping config file and generates the mapping based on it. Then, records are submitted to and indexed by ES. Indexes are created separated into segments. Segments can be on one system or across server nodes in cluster. When searches happen, the segments are searched in parallel, and then the results merged.

For more information, see: Elasticsearch from the bottom up (EuroPython 2014 - Start at 18:28 unless you want to get deep into Lucene.): https://www.youtube.com/watch?v=PpX7J-G2PEo

ES Filters versus Queries

An important distinction is between filters and queries in ES. For CR purposes, filters can be used for blocking. Queries result in a score that is assigned to the result based on how well it matches the query. For more, eee: https://logz.io/blog/elasticsearch-queries/ Note that filters are cached, and thus faster. Also, query clauses can be used as either a filter or a query. Since filters are boolean (true/false), they are not scored.

There are queries of diverse types, including Boolean, compound (chaining queries together), fuzzy matching, and other types. Queries result in a score that

is assigned to the result based on how well it matches the query. With regard to risk for the Uganda use case, we have included blocking (filters) to meet the use requirements and the required query types.

For more, see ElasticSearch Queries And the ES Query Domain Specific Language (DSL)

Last update: February 17, 2020

Security

> **Warning**
>
> It is difficult (and irresponsible) to try to explain all of the best practices in computer security. This page focuses on how security is addressed in the Open Client Registry. The information may be incomplete. Where there is more clarity or information needed, please provide feedback in an issue on GitHub so that it can be added.

This page addresses several security areas, including hardening, user authentication, node authentication, auditing, and non-production (demos, tests) configurations.

For links to information on server resource planning see the production page.

Hardening

> **Warning**
>
> Server and network hardening and production best practices are out of scope. This document only attempts to capture aspects relevant to the Client Registry.

General Server, Network, and Service Hardening

Hardening and production best practices include:

- Removing unnecessary services, software, network protocols
- Backup and recovery
- Patches
- Vulnerability scanning
- Limiting remote administration
- Managing open internal and external ports

- Auditing, logging software

See, for example, the Guide to General Server Security: Recommendations of the National Institute of Standards and Technology by Karen Scarfone, Wayne Jansen, Miles Tracy, July 2008, (NIST Special Publication 800-123).

Client Registry Platform Hardening

In addition to the above general hardening practices that should be followed, some additional areas are important for the Client Registry.

- The admin interface should only be available on a local subnet, and further restricted with user and node authentication so that it is not exposed on the WAN.
- **Close external ports**: Ports for Client Registry services should be locked down to localhost. The only external port required is for TLS with point-of-service systems that make queries to the Client Registry Service. This includes ES and HAPI FHIR. Those services can serve localhost only and effectively.
- **TLS for interservice communication**: Where Postgres and ES must communicate with other nodes TLS can be configured for ES for internode communication and Postgres for replication.
- **Disable HAPI Web Testing**: The HAPI FHIR Server Web Testing UI should be disabled. This tool allows the viewing of demographic records on the server. It is not a tool suitable for production, or if it is used, then it is restricted to a local subnet and further restricted with user and node authentication.
- **Double-check for default passwords**: Ensure no default passwords are in use for OpenHIM core and console, HAPI FHIR Server, ES, Postgres, and other services.

Authentication

User Authentication

Point-of-service systems should possess an appropriate identity provider solution built-in or externally. This also means following best practices for user authentication.

The Client Registry Service and admin interface are the only direct access to demographic data systems that should require user authentication. As this Client Registry is meant for further customization during deployment, JWT and other user authentication solutions are not provided out-of-the-box but can be added.

It is intended to support user-requested user authentication solutions as use cases are further identified.

Node Authentication

The Client Registry follows best practices outlined in the ITI-19 standard for node authentication. Only secure nodes – one with the ability to authenticate itself to other nodes and transmit data securely – should be allowed to communicate with the Client Registry.

Systems administrators should ensure that all clients must be registered and certificates assigned to them. In production, the Client Registry should act as an OpenHIM mediator which provides an extra layer of security. Clients may be registered in OpenHIM.

ATNA Logging

OpenHIM supports the Audit Trail and Node Authentication (ATNA) Integration Profile, which establishes a standard for responsibly storing audit events. It is highly recommended that the OpenHIM be configured as such but only after it is ensured that all default passwords have been changed and the OpenHIM is operating on a local subnet and thus not exposed externally. See the above hardening notes.

See the OpenHIM user guide for information on ATNA configuration.

Non-Production

In non-production settings only may self-signed certificates be created for testing and demonstrations. An example is as follows:

```
openssl req -newkey rsa:4096 -keyout dhis2_key.pem -out
dhis2_csr.pem -nodes -days 365 -subj "/CN=dhis2"
openssl x509 -req -in dhis2_csr.pem -CA ../certificates/
server_cert.pem -CAkey ../certificates/server_key.pem -out
dhis2_cert.pem -set_serial 01 -days 36500
openssl pkcs12 -export -in dhis2_cert.pem -inkey dhis2_key.pem -out
dhis2.p12
```

Last update: February 16, 2020

OpenMRS MPI Client

There is a reference implementation for 2.x and legacy OpenMRS to connect to the CR.

- 2.x support
- Legacy support

Last update: February 17, 2020

Installation (Docker)

> **Warning**
>
> The latest tag is not supported by ES.

Under construction.

Last update: February 16, 2020

Installation (Lite)

> **Warning**
>
> This guide is for demonstrations or tests only, not for production environments.

> **Note**
>
> This installation method requires familiarity with the command line.

HAPI FHIR Server CLI

For non-production environments, the HAPI maintainers provide a simple CLI-based tool to run it.

The only required dependency is Java >= 8 (1.8).

See HAPI FHIR CLI for instructions for the OS of choice.

The Client Registry requires FHIR version R4 and HAPI must be started for this version. To run HAPI:

```
hapi-fhir-cli run-server -v r4
```

The HAPI Web Testing UI is available at http://localhost:8080/ The Web Testing UI should be disabled for production. It allows the viewing of any resource on the server.

The FHIR Base URL is at http://localhost:8080/baseR4/

Visit http://localhost:8080/ to ensure HAPI is up and running or

```
curl -X GET "localhost:8080/baseR4/Patient?"
```

ElasticSearch (Optional)

By default, the configuration does not require ES but including it will enable ES-based matching.

Install and start ES for the intended OS. See the ES install instructions

The required version is >=7.5.

Once installed and started, ensure that ES is up and running:

```
curl -X GET "localhost:9200/_cat/health?v&pretty"
```

Status should be yellow for a single-node cluster.

Client Registry Service (Standalone)

Clone the repository into a directory of choice.

```
git clone https://github.com/intrahealth/client-registry.git
```

Enter the server directory, install node packages.

```
cd client-registry/server
npm install
```

Copy and edit the configuration file to your liking.

```
cp config/config_development_template.json config/
config_development.json
# edit the servers...
```

The minimum changes to start a running standalone system are:

- Change `fhirServer.baseURL` to "http://localhost:8080/baseR4/"

Run the server from inside client-registry/server:

```
node lib/app.js
```

Congratulations! Now it's time to run a query.

Last update: February 16, 2020

Installation (Production)

> **Warning**
>
> Installing and maintaining a production installation is not trivial. This installation method requires strong familiarity with the command line and expertise administering Linux environments.

The production stack consists of 10 components:

- OpenHIM core and OpenHIM admin console. Requires MongoDB and Nginx or another reverse proxy.
- Client Registry Service as an OpenHIM mediator.
- HAPI FHIR Server with a database backend.
- ElasticSearch (>=7.5)
- Client Registry UI
- Kibana Dashboard

The expected operating system is Linux for production. Only Ubuntu versions have been tested.

It is critical that systems administrators note the version compatibilities outlined below. This guide does not cover most aspects of enterprise systems administration, rather it attempts to cover this Client Registry platform. If there are key areas missing, please open an issue on GitHub.

Prerequisites

- If entities outside of your LAN are connecting to the Client Registry, you will need a public-facing domain name. This is necessary for a certificate which is required for any interactions.
- See production considerations

OpenHIM

OpenHIM supports the last 2 versions of NodeJS LTS and requires MongoDB.

- Follow the instructions to install OpenHIM core and admin console. The maintainers use the NPM PPA installation method.
- Note the important step to obtain a certificate immediately after installation. The configuration should be that any client must have a certificate and the server has a certificate (mutual TLS).
- Follow the instructions including console configuration.
- Note the important step to change the console password. It is also recommended that the console only be accessible on a local subnet and not to the WAN.

HAPI FHIR Server and Postgres

HAPI FHIR should use a database backend in production. HAPI FHIR stores the patient demographic data from queries. If the data is lost, then the Client Registry is unrecoverable.

- Follow the JPA Server information and instructions for how to customize the hapi.properties file and build the server using maven.
- The ES integration is separate from HAPI FHIR Server, so there is not need to use it as an indexer. ES only works with an old version of ES.
- Install and configure the preferred database. Postgres has been tested by the maintainers but any database should work that HAPI supports. Change default passwords on the database.
- Database replication should be encrypted.
- Confirm that HAPI accepts requests.
- The web interface for HAPI should be disabled for privacy reasons.

!! In production, Postgres should run on multiple nodes with replication. This is to ensure high availability and backups of the data.

ElasticSearch

- Follow the instructions for installation
- Systemd is the preferred system and service manager. There are commands to initiate systemd and journalctl.

!! warning ES is not production-ready when run one a single node. It is recommended to run ES on several nodes. Those nodes can also run followers of Postgres.

Client Registry UI

Under construction.

Kibana Dashboard

Under construction.

Last update: February 16, 2020

Example API Query (cURL)

Certificates (Mandatory)

A system querying the Client Registry needs a server-issued certificate or it will not be authorized to use the service.

The way that this works is that a server creates a certificate for a client. The certificate is signed by the server issuing it. The querying system then uses that certificate that has been issued to them in their requests. The server's public key is used by the querying system to verify that the certificate being sent is how the server verifies that the certificate was created by them.

There is a set of generated certificates for testing and demonstrations. They are not appropriate for production.

A Simple CLI Query

Inside /client-registry/server directory, a cURL query using the provided example JSON file would be:

```
curl --cert sampleclientcertificates/openmrs.p12 --cert-type p12 --
cacert certificates/server_cert.pem -d @/Users/richard/src/
github.com/openhie/client-registry/DemoData/patient1_openmrs.json -
H "Content-Type: application/json" -XPOST https://localhost:3000/
addPatient
```

Should result in a successful result in stdout:

```
info: Received a request to add new patient
{"timestamp":"2020-01-28 14:29:20"}
info: Searching to check if the patient exists
{"timestamp":"2020-01-28 14:29:20"}
info: Getting http://localhost:8080/baseR4/Patient?identifier=431287
from server {"timestamp":"2020-01-28 14:29:20"}
info: Patient [{"system":"http://clientregistry.org/
openmrs","value":"431287"},{"system":"http://
system1.org","value":"12349","period":
```

{"start":"2001-05-06"},"assigner":{"display":"test Org"}}] doesnt exist, adding to the database {"timestamp":"2020-01-28 14:29:20"}

Last update: February 16, 2020

Example API Query (Python)

> **Note**
>
> This example is available in the Jupyter notebook at: github.com/intrahealth/client-registry/examples/simple_query.ipynb

This is a simple query in Python exactly like the cURL query:

```
curl --cert sampleclientcertificates/openmrs.p12 --cert-type p12 --
cacert certificates/server_cert.pem -d @/Users/richard/src/
github.com/openhie/client-registry/DemoData/patient1_openmrs.json -
H "Content-Type: application/json" -XPOST https://localhost:3000/
addPatient
```

The only requirement is a version of the requests package that has been modified to take p12 certs.

```
pip3 install requests_pkcs12
```

Import the required modules.

```
from pathlib import Path
# import requests modded dor pkcs12
from requests_pkcs12 import get, post
```

This example assumes running from `github.com/intrahealth/client-registry/examples` . Change paths as required. Using the Path module ensures that the path resolution works in Mac, Linux, Windows.

```
clientcert_folder = Path("../server/sampleclientcertificates")
clientcert = clientcert_folder / "openmrs.p12"

servercert_folder = Path("../server/certificates")
servercert = servercert_folder / "server_cert.pem"

payload_folder = Path("../DemoData/")
```

```
payload_bytes = payload_folder / "patient1_openmrs.json"
payload = open(payload_bytes)
```

Define headers and initiated the POST request.

```
headers = {'Content-Type': 'application/json'}
response = post("https://localhost:3000/addPatient",
headers=headers, data=payload,
                pkcs12_filename=clientcert,
                pkcs12_password='',
                verify=servercert)
print(response)
```

Last update: February 16, 2020

Server Configuration

Often there are many records of the same person but in many people in different systems. The purpose of the Client Registry is to link patients in different systems, but not to transfer any data, neither clinical records nor demographic data.

> **Caution**
>
> The Client Registry does not store clinical information. Having the Client Registry enables the ability to create a Shared Health Record in the future.

The Client Registry stores the patient demographic data submitted to it in queries. The Client Registry stores demographic data at least in the HAPI FHIR Server, which can have any database backend an implementer chooses to use.

ElasticSearch (ES) is an optional search engine, and requires configuration. ES can also store patient data fields selectably.

JSON files are used to configure the system. Later iterations will support environment variables and a graphical interface. See https://github.com/openhie/client-registry/tree/master/server/config for example configuration files discussed here.

Deciding Between a Standalone or Mediator Configuration

A central application is the Client Registry Service, as distinct from the larger Client Registry platform. There are two options for running the application, as an OpenHIM mediator or as standalone application.

Choose running the app standalone when:

- For testing, demonstration, or development environments.

- There are few clients that will connect to managing client authentication and roles will not be a burden.
- There is no need for an additional layer of auditing.

Choose running the app as a mediator when:

- For production. The central application should be run as a mediator registered in OpenHIM.
- There are many clients that will need to connect.
- There is a need to audit transactions.
- There is an existing health information exchange layer or OpenHIM.
- One advantage of using the OpenHIM interface is the ability to change settings like the FHIR server.

Security and Privacy

Many configuration options relate to privacy and security. These steps are critical to address. See the security page

Whether in standalone or as a mediator, the Client Registry must interact only with known, trusted clients with TLS certificates. Clients must be registered and certificates assigned to them.

In standalone mode, the server runs TLS by default, and requires signed certificates. Client certificate needs can be turned off in OpenHIM when running as a mediator and this feature must be regularly audited to ensure security.

Connecting Services

The default ports are as follows:

- **3000**: Client Registry Service
- **9200**: ElasticSearch (closed to external)
- **8080**: HAPI FHIR Server (closed to external)

In `server/config/config_development_template.json` there is a template for configuration.

Link to file

Contents of `server/config/config_development_template.json`

```
{
  "app": {
    "port": 3000,
    "installed": false
  },
  "mediator": {
    "api": {
      "username": "root@openhim.org",
      "password": "openhim-password",
      "apiURL": "https://localhost:8080",
      "trustSelfSigned": true,
      "urn": ""
    },
    "register": false
  },
  "fhirServer": {
    "baseURL": "http://localhost:8080/clientregistry/fhir",
    "username": "hapi",
    "password": "hapi"
  },
  "elastic": {
    "server": "http://localhost:9200",
    "username": "",
    "password": "",
    "max_compilations_rate": "10000/1m",
    "index": "patients"
  },
  "structureDefinition": {
    "reportRelationship": "patientreport"
  },
  "matching": {
    "tool": "mediator"
  },
  "systems": {
    "openmrs": {
      "uri": "http://clientregistry.org/openmrs"
    },
    "dhis2": {
      "uri": "http://clientregistry.org/dhis2"
    },
```

```
    "lims": {
      "uri": "http://clientregistry.org/lims"
    },
    "brokenMatch": {
      "uri": "http://ihris.org/CR/brokenMatch"
    }
  },
  "sync": {
    "lastFHIR2ESSync": "1970-01-01T00:00:06"
  },
  "__comments": {
    "matching.tool": "this tells if the app should use mediator
algorithms or elasticsearch algorithms for matching, two options
mediator and elasticsearch"
  }
}
```

General App Configuration

`app.port` is the port the application will run on.

`app.installed` can be left to True. This tells the Client Registry Service to load structure definitions into HAPI FHIR Server, otherwise it will not.

Mediator App Configuration

`mediator.register` to true if the application will run as a mediator. Or, to false if the app will run as standalone.

`mediator.api.xx` settings are only if running as a mediator.

`mediator.api.username | password` must be different. The existing settings are defaults and must be changed when configuring the OpenHIM.

`mediator.api.trustSelfSigned` should be set to false in production or any sensitive environment. True is only for demonstrations.

FHIR Server

The currently supported FHIR version is R4.

`fhirServer.baseURL` is the default. Note that it may change depending on the way HAPI is installed. It may, for example, default to a baseURL of http://localhost:8080/baseR4/.

`fhirServer.username | password` must be changed from defaults in HAPI.

ElasticSearch Configuration

For ES, the relationship between patient resources in FHIR and what fields are synchronized in ES must be explicitly defined. This is termed the Report Relationship mapping. One must define what resource to be used (patient) and what fields need to be available in ES. After this, the Client Registry reads these fields, and populates ES with the information.

In `resources/Relationships/PatientRelationship.json` there is a template for configuration.

Link to file

Contents of `resources/Relationships/PatientRelationship.json`

```
{
  "resourceType": "Basic",
  "id": "patientreport",
  "meta": {
    "versionId": "1",
    "lastUpdated": "2019-07-30T07:34:24.098+00:00",
    "profile": [
      "http://ihris.org/fhir/StructureDefinition/iHRISRelationship"
    ]
  },
  "extension": [{
    "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportDetails",
    "extension": [{
      "url": "label",
      "valueString": "Patient Report"
    }, {
      "url": "name",
      "valueString": "patients"
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportElement",
```

```
      "extension": [{
        "url": "label",
        "valueString": "gender"
      }, {
        "url": "name",
        "valueString": "gender"
      }]
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportElement",
      "extension": [{
        "url": "label",
        "valueString": "birthDate"
      }, {
        "url": "name",
        "valueString": "birthDate"
      }]
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportElement",
      "extension": [{
        "url": "label",
        "valueString": "given"
      }, {
        "url": "name",
        "valueString": "name.where(use='official').last().given"
      }]
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportElement",
      "extension": [{
        "url": "label",
        "valueString": "family"
      }, {
        "url": "name",
        "valueString": "name.where(use='official').last().family"
      }]
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
iHRISReportElement",
      "extension": [{
        "url": "label",
        "valueString": "fullname"
      }, {
        "url": "name",
        "valueString": "name.where(use='official').last().text"
      }]
    }, {
      "url": "http://ihris.org/fhir/StructureDefinition/
```

```
iHRISReportElement",
      "extension": [{
        "url": "label",
        "valueString": "phone"
      }, {
        "url": "name",
        "valueString": "telecom.where(system='phone').value"
      }]
    }]
  }],
  "code": {
    "coding": [{
      "system": "http://ihris.org/fhir/ValueSet/ihris-resource",
      "code": "iHRISRelationship"
    }],
    "text": "iHRISRelationship"
  },
  "subject": {
    "reference": "StructureDefinition/Patient"
  }
}
```

OpenHIM Mediator JSON Configuration

If using OpenHIM, it must be configured for proper clients and roles to accept and forward requests from the Client Registry. An example export of a working JSON configuration that can be imported for development purposes is available.

Link to file

Contents of `server/config/mediator.json`

```
    {
      "urn": "urn:uuid:4bc42b2f-b5a8-473d-8207-5dd5c61f0c4a",
      "version": "0.0.1",
      "name": "Client Registry",
      "description": "Uganda Client Registry",
      "config": {
        "fhirServer": {
          "username": "hapi",
          "password": "hapi",
          "baseURL": "http://localhost:8080/hapi/fhir"
        },
        "elastic": {
          "server": "http://localhost:9200",
```

```
          "username": "",
          "password": "",
          "max_compilations_rate": "10000/1m",
          "index": "patients"
        },
        "matching": {
          "tool": "elasticsearch"
        }
      },
      "configDefs": [{
        "param": "fhirServer",
        "displayName": "FHIR Server",
        "description": "FHIR Server Configuration Details",
        "type": "struct",
        "template": [{
            "type": "string",
            "description": "The base URL (e.g. http://localhost:
8080/hapi/fhir)",
            "displayName": "Base URL",
            "param": "baseURL"
          },
          {
            "type": "string",
            "description": "Username required to access FHIR
server",
            "displayName": "Username",
            "param": "username"
          },
          {
            "type": "password",
            "description": "Password required to access FHIR
server",
            "displayName": "Password",
            "param": "password"
          }
        ],
        "values": []
      }, {
        "param": "elastic",
        "displayName": "Elasticsearch Server",
        "description": "Elasticsearch Server Configuration Details",
        "type": "struct",
        "template": [{
            "type": "string",
            "description": "The base URL (e.g. http://localhost:
9200)",
            "displayName": "Base URL",
            "param": "server"
          },
```

```
          {
            "type": "string",
            "description": "Username required to access
elasticsearch server",
            "displayName": "Username",
            "param": "username"
          },
          {
            "type": "password",
            "description": "Password required to access
elasticsearch server",
            "displayName": "Password",
            "param": "password"
          }, {
            "type": "string",
            "description": "Number of requests to compile per
minute",
            "displayName": "Maximum Compilations Rate",
            "param": "max_compilations_rate"
          }, {
            "type": "string",
            "description": "index to use for data storage",
            "displayName": "Index Name",
            "param": "index"
          }
        ],
        "values": []
      }, {
        "param": "matching",
        "displayName": "FHIR Server",
        "description": "FHIR Server Configuration Details",
        "type": "struct",
        "template": [{
          "type": "option",
          "values": ["mediator", "elasticsearch"],
          "description": "Tool to Use for Matching",
          "displayName": "Tool to Use for Matching",
          "param": "tool"
        }],
        "values": []
      }],
      "defaultChannelConfig": [{
        "requestBody": true,
        "responseBody": true,
        "name": "Add Patients",
        "description": "Post a new patient into the client
registry",
        "urlPattern": "/addPatient",
        "matchContentRegex": null,
```

```
        "matchContentXpath": null,
        "matchContentValue": null,
        "matchContentJson": null,
        "pollingSchedule": null,
        "tcpHost": null,
        "tcpPort": null,
        "autoRetryPeriodMinutes": 60,
        "autoRetryEnabled": false,
        "rewriteUrlsConfig": [],
        "addAutoRewriteRules": true,
        "rewriteUrls": false,
        "status": "enabled",
        "alerts": [],
        "txRerunAcl": [],
        "txViewFullAcl": [],
        "txViewAcl": [],
        "properties": [],
        "matchContentTypes": [],
        "routes": [{
          "name": "Add Patient",
          "secured": false,
          "host": "localhost",
          "port": 3000,
          "path": "/addPatient",
          "pathTransform": "",
          "primary": true,
          "username": "",
          "password": "",
          "forwardAuthHeader": false,
          "status": "enabled",
          "type": "http"
        }],
        "authType": "public",
        "whitelist": [],
        "allow": [],
        "type": "http",
        "methods": [
          "POST"
        ]
      }],
      "endpoints": [{
        "name": "Activate Client Registry",
        "host": "localhost",
        "path": "/addPatient",
        "port": 3000,
        "primary": true,
        "forwardAuthHeader": false,
        "status": "enabled",
        "type": "http"
```

```
      }],
      "_uptime": 2201.945,
      "_lastHeartbeat": "2017-12-15T03:47:03.365Z",
      "_configModifiedTS": "2017-12-15T02:52:49.054Z"
    }
```

Last update: February 16, 2020

Configure Benchmarks

Last update: February 16, 2020

Production Considerations

This page is under construction.

IT Resource Planning

Benchmarking will be completed in future phases to make recommendations for medium to heavy workloads. The below resource suggestions should be revised based on benchmarking for the particular context into which the Client Registry is being deployed.

Servers

For an MVP in a production environment where potential data loss is acceptable, a single large server can be used. ES has high memory requirements.

CPU

- Local development: PC with multiple cores
- Production: Use 2-8 cores.

Memory

Memory usage depends on the number of records and the performance required.

Local development on a PC: 8GB of RAM should be available

Production, at minimum: 32GB with 24GB free for the Client Registry is recommended for light loads if using one VM.

- 16GB minimum for ElasticSearch with 32GB preferred or 64GB for high volume: Follow the guidelines provided by the maintainers here.
- 8GB for OpenHIM, mediator, Postgres, and HAPI FHIR Server.

Disk Space

This depends heavily on the workload. Expect 200GB at minimum per node.

Last update: February 16, 2020

Backup and Recovery

Backup

The primary datastore is the database configured for HAPI FHIR Server. This means that while an ES cluster should be backed-up, the ES index can be rebuilt from HAPI.

Recovery

Last update: February 16, 2020

Troubleshooting

Last update: February 16, 2020

Building Documentation

This documentation is built using MkDocs and the Material for MkDocs theme.

PDF export is done using mkdocs-pdf-export-plugin. All configuration information is in mkdocs.yaml in the repository. Note that at some future time the docs may be migrated into the main client registry repository.

Edits to docs are made in the master branch of the client registry repository docs repo.

After docs are edited, they are pushed to origin master, and then the `mkdocs gh-deploy` is run on the command line. This pushes into the gh-pages branch on GitHub. Only master is ever edited. The gh-pages is only modified by the CLI.

Last update: February 16, 2020

Contributing

There may be many areas of potential contribution as the Client Registry is not one application, it's several and can be more than that in your use case.

It's recommended that you identify the specific feature or use case that needs support and

For a quick question, reach out on the iHRIS Slack team. Sign up here

For a bug or feature, reach out to the relevant repository to share the information. See the developer page for links to the different applications.

For a broader discussion with others interested and with a background in Client Registry implementation science, please join the OpenHIE Client Registry Community calls and get involved.

Last update: January 22, 2020

License

License

CR Service

The CR Service is under a permissive license.

HAPI FHIR Server

HAPI is licensed under the Apache 2.0 license. See: https://hapifhir.io/hapi-fhir/license.html.

ElasticSearch

ElasticSearch is Apache 2.0 and primarily supported by Elastic.co who offer open and open plus proprietary releases and a stack of apps based on ElasticSearch (like the Kibana dashboarding platform).

This CR implementation only uses core, open features, not enterprise features. Netflix and Amazon Web Services also have a distribution of ElasticSearch that is 100% open source and that they have added open source enterprise features to like authentication. Developers who want to test other distributions and features of ElasticSearch are welcome to do so and encouraged to share their experiences with the maintainers.

Open Distro for ElasticSearch

Last update: February 17, 2020

About

OpenCR was developed by IntraHealth International with support from PEPFAR through the USAID MEASURE Evaluation Project. Technical direction was provided by CDC.

Last update: February 20, 2020