OpenHIM Documentation

Release 1.3.0

Jembi Health Systems

June 07, 2016

Contents

Contents:

CHAPTER 1

About the OpenHIM

OpenHIM stands for the Open Health Information Mediator. The OpenHIM is an interoperability layer: a software component that eases integration between disparate information systems by connecting client and infrastructure components together. Its role is to provide a facade to client systems - providing a single point-of-control for defining web service APIs and adapting and orchestrating requests between infrastructure services. The OpenHIM also provides security, client management and transaction monitoring.

The OpenHIM was initially developed as part of the Rwandan Health Enterprise Architecture project in collaboration with the University of KwaZulu-Natal and was further developed as part of the OpenHIE initiative, where it serves as an interoperability layer reference implementation. The OpenHIM tool is also supported through various implementation projects that continue to support its growth to meet real world needs and project requirements.

1.1 Ok. But, what does the HIM actually do?

The OpenHIM is split into two logical parts:

1. The core
2. Optional mediators

The OpenHIM-core performs common functions that are useful for a SOA environment. It provides an interface that point of service application (clients) are able to contact in order to reach the services provided in the SOA. You can think of this interface as a reverse proxy for your applications but with some special features. These features include:

- Access control - defaults to mutual TLS (client and server certificates) but can also be set to basic auth
- Every request and response is stored for accountability
- Metrics are calculated to give an indication of how the SOA is running
- Support for HTTP request, plain sockets or MLLP sockets
- Multi-casting of requests to multiple endpoints

The mediators are optional, pluggable services that can add extended specific functionality to the OpenHIM. These are often used for the following types of use cases:

- Message format adaptation - this allows the transformation of messages received in a certain format into another format (eg. HL7 v2 to HL7 v3 or MHD to XDS.b).
- Message orchestration - this allows the execution of a business process that may need to call out to other service endpoint on other system. (eg. Enriching a message with a client's unique identifier retrieved from a client registry).

The OpenHIM-core provides a framework to add and mange your own implementation specific mediators in the system.

The OpenHIM also comes with an admin console which provides a user friendly interface for configuring and managing an OpenHIM instance.

If you want to install and implement the OpenHIM, see the getting started section and the user guide for information on how to get setup and how to configure the system using the OpenHIM console.

1.2 Roadmap

To see the roadmap for the OpenHIM you can view the issues list for core and for console. When a release is being worked on we try to maintain a milestone for that release. You can see core milestones here and console milestones here. To view what went into our latest releases, have a look at our releases pages. Core releases and console releases.

1.3 Funders

UNITED STATES PRESIDENT'S EMERGENCY PLAN FOR AIDS RELIEF

SOUTH AFRICANS AND AMERICANS
IN PARTNERSHIP TO FIGHT HIV/AIDS

PEPFAR

CDC
CENTERS FOR DISEASE
CONTROL AND PREVENTION

IDRC | CRDI
International Development Research Centre
Centre de recherches pour le développement international
Canadä

Rockefeller Foundation
Innovation for the Next 100 Years

1.4 Development Partners

1.5 Other Partners

eCGroup

CHAPTER 2

Getting started

To get started we will show you how to install the OpenHIM along with the admin console for easy configuration.

If you are on Ubuntu installing the OpenHIM is very easy as we provide a debian package in the openhie PPA. Just execute the following commands:

```
$ sudo add-apt-repository ppa:openhie/release
$ sudo apt-get update
$ sudo apt-get install openhim-core-js openhim-console
```

When installing the console, it will ask you for the host and port of the OpenHIM-core server. Make sure you provide the **public** hostname where the OpenHIM-core server will be accessible (localhost is fine if you are testing and just want access on your local machine). You can run `sudo dpkg-reconfigure openhim-console` at any time to specify a new OpenHIM-core host and port.

These packages will install the OpenHIM-core using NPM for the OpenHIM user, add the OpenHIM-core as a service and install the console to nginx. You can find the core log file here `/var/log/upstart/openhim-core.log` and may stop and start the OpenHIM-core with `sudo start openhim-core` or `sudo stop openhim-core`.

If you don't have ubuntu or want to install manually, follow the steps below.

2.1 Installing the OpenHIM-core

1. Install the latest stable Node.js 0.12.0 or greater.
2. Install and start MongoDB 2.6 or greater.
3. Install the OpenHIM-core package globally: `npm install openhim-core -g`, this will also install an OpenHIM-core binary to your PATH.
4. Start the server by executing `openhim-core` from anywhere.

To make use of your own custom configurations you can copy the default.json config file and override the default settings:

```
wget https://raw.githubusercontent.com/jembi/openhim-core-js/master/config/default.json
# edit default.json, then
openhim-core --conf=path/to/default.json
```

For more information about the config options, click here.

Note: one of the first things that you should do once the OpenHIM is up and running is setup a properly signed TLS certificate. You can do this later through the OpenHIM console under 'Certificates' on the sidebar.

2.2 Installing the OpenHIM admin console

First ensure that you have the OpenHIM-core server up and running. The console communicates with the OpenHIM-core via its API to pull and display data.

Next, you need to pull down the latest release of the web app and deploy it to a web server (replace the X's in the below command to the latest release):

```
wget https://github.com/jembi/openhim-console/releases/download/vX.X.X/openhim-console-vX.X.X.tar.gz
tar -vxzf openhim-console-vX.X.X.tar.gz --directory /var/www/
```

Next, and this step is *vital*, you need to configure the console to point to your OpenHIM-core server. Locate `config/default.js` in the folder you extracted the OpenHIM console to and edit it as follows:

```
{
  "protocol": "https",
  "host": "localhost",  // change this to the hostname for your OpenHIM-core server (This hostname _M
  "port": 8080,         // change this to the API port of the OpenHIM-core server, default is 8080 (T
  "title": "OpenHIM Admin Console", // You may change this to customise the title of the OpenHIM-cons
  "footerTitle": "OpenHIM Administration Console", // You may change this to customise the footer of
  "footerPoweredBy": "<a href='http://openhim.org/' target='_blank'>Powered by OpenHIM</a>",
  "loginBanner": ""     // add text here that you want to appear on the login screen, if any.
}
```

Now, navigate to your web server and you should see the OpenHIM-console load (eg. `http://localhost/`) and login. The default username and password are:

- username: `root@openhim.org`
- password: `openhim-password`

You will be prompted to change this after your first successful login.

Note: You will have problems logging in if your OpenHIM server is still setup to use a self-signed certificate (the default). To get around this you can use the following workaround (the proper way to solve this is to upload a proper certificate into the OpenHIM-core):

> Visit the following link: `https://localhost:8080/authenticate/root@openhim.org` in Chrome. Make sure you are visiting this link from the system that is running the OpenHIM-core. Otherwise, replace `localhost` and `8080` with the appropriate OpenHIM-core server hostname and API port. You should see a message saying "Your connection is not private". Click "Advanced" and then click "Proceed". Once you have done this, you should see some JSON text displayed on the screen, you can ignore this and close the page. This will ignore the fact that the certificate is self-signed. Now, you should be able to go back to the Console login page and login. This problem will occur every now and then until you load a properly signed certificate into the OpenHIM-core server.

You now have the OpenHIM with admin console successfully up and running. From here you may want to checkout our tutorials or continue on to the user guide to learn more about how to configure your instance.

CHAPTER 3

Tutorial

3.1 Tutorial start

Before getting started with the tutorials, please ensure that

- the OpenHIM has been installed (see here for tips on proceeding with this if not done so already)
- the tutorial services are set up and running.

The tutorial services consist of three mock health registries that we'll be routing to and retrieving data from during the tutorials. They provide simple endpoints, hosting mock patients, their health records and healthcare providers. Once you've downloaded the services, you will need to open up three terminals so that each service can be started individually. To do so, follow the below steps:

- Unzip the folder and open up three terminal windows
- Navigate to where you have unzipped your OpenHIM Tutorial Services
- Make sure that all node dependencies are installed

```
$ npm install
```

For each of the services you will need to run following command in a new terminal:

```
$ node health-record-service.js
```

```
$ node client-service.js
```

```
$ node healthcare-worker-service.js
```

The service will indicate that it is running on a specific port. You can test these services by executing the following CURL commands and the below results should be displayed:

```
$ curl http://localhost:3444/encounters/1
```

```
{
  "patientId": 1,
  "providerId": 1,
  "encounterType": "Physical Examination",
  "encounterDate": "20131023",
  "observations": [
    {
      "obsType": "Weight",
      "obsValue": "50",
      "obsUnit": "kg"
    },
    {
      "obsType": "Height",
      "obsValue": "160",
      "obsUnit": "cm"
    },
    {
      "obsType": "Systolic Blood Pressure",
      "obsValue": "120",
      "obsUnit": "mmHg"
    },
    {
      "obsType": "Diastolic Blood Pressure",
      "obsValue": "80",
      "obsUnit": "mmHg"
    },
    {
      "obsType": "Heartrate",
      "obsValue": "90",
      "obsUnit": "bpm"
    },
    {
      "obsType": "Temperature",
      "obsValue": "37",
      "obsUnit": "C"
    }
  ]
}
```

```
$ curl http://localhost:3445/patient/1
```

```
{
  "patientId": 1,
  "familyName": "Patient",
  "givenName": "Sally",
  "gender": "F",
  "phoneNumber": "0731234567"
}
```

```
$ curl http://localhost:3446/providers/1
```

```
{
  "providerId": 1,
  "familyName": "Doctor",
  "givenName": "Dennis",
  "title": "Dr"
}
```

Once you have these running you are ready to continute to the next tutorial.

3.2 Creating your first channel

This tutorial assumes that you've set up the required dependencies, including setting up the OpenHIM, and that you are ready to create your first channel. In this tutorial we will look at setting up a channel on the HIM that will allow you to route data from a demo service. You will first need to create a **Client** that gets authenticated when requests are made. First, access the HIM Console on your browser, and login using the appropriate credentials. Next, go through to the **Clients** section (**Sidebar -> Clients**), create a new **Client** with the following details and save:

- ClientID: **tut**

- Client Name: **OpenHIM Tutorial Client**
- Domain: **openhim.org**
- Roles: **tut**
- Password: **tut**

For this tutorial, we will be using **Basic auth** for authenticating the client. The OpenHIM supports both basic authentication and mutual TLS. With this in mind, ensure that your HIM Core instance is running with Basic auth enabled. To do so, and if not already done, create a copy of the default config file and ensure that **enableBasicAuthentication** is **true** (the default). You can also override any other config if required and startup core using:

```
$ openhim-core --conf=myconfig.json
```

Next we need to create a **Channel** which will route our request. Lets go through to the **Channels** section (**Sidebar -> Channels**) and create a new **Channel** with the following details and save:

- Basic Info:
  - Name: **Tutorial Channel**
  - URL Pattern: **/encounters/.***
- Access Control:
  - Allowed roles and clients: **tut (NB! This is the Client's roles that we created previously)**
- Routes -> Add new Route:
  - Name: **Tutorial Route**
  - Primary: **True**
  - Type: **HTTP**
  - Secured: **Not Secured**
  - Host: **localhost**
  - Port: **3444**

You can use the green save button to store the route to the channel. You may also need to change **localhost** to an appropriate value, depending on your setup and the locations of your various services. This configuration allows us to create a channel that routes data from a client to the mock health record service. The URL pattern field allows us to define a regular expression for matching incoming requests to this specific channel. In this case, we're telling the HIM that any requests that match the pattern **/encounters/{anything}**, belong to this tutorial channel. The HIM will then route those requests to **http://localhost:3444/encounters/{anything}**. Note that the health records service itself is unsecured, but that security is enabled for the HIM core itself. The HIM is therefore providing a security layer for our unsecured service. Now that our Client and Channel has been created, we're ready to send through a transaction! We will be doing this using a CURL command:

```
$ curl -k -u tut:tut https://localhost:5000/encounters/1
```

If you get an **Internal Server Error** then make sure that your service is running and that **Tutorial Channel** is able to route to it. Update your channel route to use your inet addr instead of localhost Example: Channel -> route -> Host: 192.168.1.10

If you have followed Tutorial 1 correctly then your transactions should have routed through the OpenHIM and reached the Health Record Service successfully. You should see a JSON object printed on your terminal with the Health Record result. You should also be able to see the transaction's log in the HIM console. Here you will see the **/encounters/1** request was successful and the response body contains the returned JSON object.

3.3 Creating a passthrough mediator

This tutorial assumes that you have successfully completed Tutorial 1 which yielded a valid Health Record in the response body.

In the previous tutorial we created a basic channel for routing requests from the mock health record service. This is what we refer to as a **pass-through channel** - a channel that simply routes data "as is" from and to services. But we often want to do more than just pass through data unmodified. We may want to alter a request's format or enrich the request with extra information. This is where mediators come into play. A mediator is a light-weight HIM **micro-service** that performs operations on requests.

In the previous tutorial our channel returned a basic JSON health record for a patient. But wouldn't it be more useful for this tutorial if you could look at the data in an easy to read format on your web browser rather than using CURL? Or if the patient's name was included, rather than just an identifier? With a mediator we can do just that, and in this and the following tutorials we will look into creating a mediator that will convert our health record data to HTML (Note: you could convert to any data format that you choose, however, for the purposes of this tutorial we are using HTML as an example), so that we can view it easily in a web browser, and also enrich the message with patient and healthcare provider demographic information. Before we can get started though, we need to scaffold a basic mediator that we can use for these purposes, and this is what we'll be looking at in this tutorial.

If you want more detailed information on mediators and their related concepts, you can read up on them over here. You can create your mediator from scratch if you like by following these guidelines, or you can use our Yeoman Mediator Generator to scaffold your mediator in a few easy steps. We will be taking the Yeoman approach.

Install Yeoman Globally:

```
$ npm install -g yo
```

make sure yeoman is installed correctly by checking the version. execute the following command.

```
$ yo -v
```

Currently we have support for two diffrent generators. One for scafolding a node.js mediator and another for scafolding a Java mediator. You may pick which ever language you are most comfortable with.

3.3.1 NodeJS Mediator

The below steps will guide you on how to create a new NodeJS Mediator through Yeoman. First, lets create a directory where our Mediator will be created:

```
$ mkdir tutorialmediator && cd tutorialmediator
```

To create a scaffolded Mediator through Yeoman you will need to download the npm **generator-mediator-js** module. execute the following command:

```
$ npm install -g generator-mediator-js
```

Now lets create our scaffold Mediator. There are two ways to execute the yo command. Either way is fine. Bring up the Yeoman menu with all generators listed:

```
$ yo
```

Or start the process for creating a Mediator:

```
$ yo mediator-js
```

Fill out the questions that are asked by Yeoman. These are used to create your configuration.

- What is your Mediator's name?: **Tutorial Mediator**
- What does your Mediator do?: **This is the mediator being used in the tutorial**

- Under what port number should the mediator run?: **4000**
- What is your primary route path? [keep this blank]

You have successfully created your scaffolded Mediator! NB! the mediator isn't ready to be launched yet, we still need to make a few changes. **NB! Remember to install your dependencies**

```
$ npm install
```

Lets make sure all out config files have the correct values. Open up **app/config/config.json** and supply your OpenHIM config details and save:

```
"api": {
  "username": "root@openhim.org",
  "password": "openhim-password" [Please make sure you supply your updated password],
  "apiURL": "https://localhost:8080"
}
```

Open up **app/config/mediator.json** and supply your OpenHIM config details as follows. The details in this file will be used to register your mediator with and setup some useful configuration in the OpenHIM core.

```
{
  ...
  "defaultChannelConfig": [
    {
      ...
      "urlPattern": "/encounters/.*",
      ...
      "allow": ["tut"],
      ...
    }
  ],
  ...
}
```

Once the config has been supplied correctly we should have a Mediator that can be registered with the OpenHIM-core successfully. This registration will actually create a default channel for the mediator using the config that we have just supplied above. This allows your mediator to create its own channel so that a user doesn't have to do this via the OpenHIM console unless they want to change the defaults that we set. You can also supply endpoints in this config file, these will show up in the OpenHIM console so that you may easily connect to mediators. You will see this has already been filled out for you! Next, open up **app/index.js**. We will be creating a new endpoint which our Mediator will be listening on. Whenever that endpoint gets a request we will route the request onto our Health Record service.

First, let's setup the endpoint. Update the default endpoint that was created for you with the generator to listen on '/encounters/:id' just like the mock server does (this doesn't have to be the same, but it makes it easier for us).

```
// listen for requests coming through on /encounters/:id
app.get('/encounters/:id', function (req, res) {

})
```

Inside the endpoint we created above we will make a request to the service we are trying to reach. In this case it is the Health Record service. To make this easier we will use a module called **needle**. First we need to add our **needle** module to our dependencies:

```
$ npm install needle --save
```

We also need to make sure that we require our **needle** module at the top of the script:

```
...
var app = express()
var needle = require('needle');
```

Now we need to send our **needle** request to our Health Record service, this should be done inside the function we created above and should wrap the existing code that was generated.

```
//send HTTP request to Health Record service
needle.get('http://localhost:3444/encounters/'+req.params.id, function(err, resp) {
  ... existing generated code ...
});
```

NB! make sure that you request path is correct. IE the endpoint is reachable We will now be working inside the request we are making to the Health Record Service. Add an error exception log message

```
// check if any errors occurred
if (err){
  console.log(err)
  return;
}
```

Mediator are able to communicate metadata back to the OpenHIM-core. This metadata includes details about the requests that the mediator made to other serves and the responses that it received. Each request that the mediator makes to other services are called orchestrations. We need to build up a orchestration object for the request that we are sending to the Health Record service. Below the error handling that we just added update the context object and orchestration data that the generator pre-populated for you with the following data:

```
/* ######################################### */
/* ##### Create Initial Orchestration  ##### */
/* ######################################### */

// context object to store json objects
var ctxObject = {};
ctxObject['encounter'] = resp.body;

//Capture 'encounter' orchestration data
orchestrationsResults = [];
orchestrationsResults.push({
  name: 'Get Encounter',
  request: {
    path : req.path,
    headers: req.headers,
    querystring: req.originalUrl.replace( req.path, "" ),
    body: req.body,
    method: req.method,
    timestamp: new Date().getTime()
  },
  response: {
    status: resp.statusCode,
    body: JSON.stringify(resp.body, null, 4),
    timestamp: new Date().getTime()
  }
});
```

This data will be used in the OpenHIM console so show what this mediator did to the message. Each step that a mediator performs to process a message, be it making a request to an external service or just transforming the message to another format, is called an orchestration. Here we have just created a single orchestration as the mediator doesn't do anything except pass the message along. The request and response data can be easily set depending on what we send and receive from the mock service. Next, we will edit the response object to tell the OpenHIM core what we want to return to the client. We will also attach the orchestration that we created above. This is the response that we will build up and return the the OpenHIM core:

```
/* ###################################### */
/* ##### Construct Response Object  ##### */
/* ###################################### */

var urn = mediatorConfig.urn;
var status = 'Successful';
var response = {
  status: resp.statusCode,
  headers: {
    'content-type': 'application/json'
  },
  body: JSON.stringify(resp.body, null, 4),
  timestamp: new Date().getTime()
};

// construct property data to be returned - this can be anything interesting that you want to make av
var properties = {};
properties[ctxObject.encounter.observations[0].obsType] = ctxObject.encounter.observations[0].obsValu
properties[ctxObject.encounter.observations[1].obsType] = ctxObject.encounter.observations[1].obsValu
properties[ctxObject.encounter.observations[2].obsType] = ctxObject.encounter.observations[2].obsValu
properties[ctxObject.encounter.observations[3].obsType] = ctxObject.encounter.observations[3].obsValu
properties[ctxObject.encounter.observations[4].obsType] = ctxObject.encounter.observations[4].obsValu
properties[ctxObject.encounter.observations[5].obsType] = ctxObject.encounter.observations[5].obsValu

// construct returnObject to be returned
var returnObject = {
  "x-mediator-urn": urn,
  "status": status,
  "response": response,
  "orchestrations": orchestrationsResults,
  "properties": properties
}
```

Once we have our return object setup correctly we will send the response back to the OpenHIM core. The generator should have already created this code for you.

```
// set content type header so that OpenHIM knows how to handle the response
res.set('Content-Type', 'application/json+openhim');
res.send(returnObject);
```

Your OpenHIM Mediator should be completed now. Lets fire-up the server and see if it registered correctly. You can start the server with:

```
$ grunt serve
```

You should see the below message if your Mediator started/registered successfully:

```
Attempting to create/update mediator Mediator has been successfully
created/updated.
```

Navigate to the **Mediators** section on OpenHIM console to see your Mediator‘

3.3.2 Java Mediator

The below steps will guide you on how to scaffold a new Java Mediator through Yeoman. First, lets create a directory where our Mediator will be created. Create a folder called **tutorialmediator**

```
$ mkdir tutorialmediator cd tutorialmediator
```

To create a scaffolded Mediator through Yeoman you will need to download the npm **generator-mediator-java** module. execute the following command:

```
$ npm install -g generator-mediator-java
```

Now lets create our scaffold Mediator. There are two ways to execute the yo command. Either way is fine. Bring up the Yeoman menu with all generators listed:

```
$ yo
```

Or start the process for creating a Mediator:

```
$ yo mediator-java
```

Fill out the questions that are asked by Yeoman. These are used to create your configuration.

- What is your Mediator's name?: **Tutorial Mediator**
- What does your Mediator do?: **This is the mediator being used in the tutorial**
- What is your group ID?: **tutorial**
- What artifact ID do you want to use?: **tutorial-mediator**
- What package do you want to use for the source code?: **tutorial**
- Under what port number should the mediator run?: **4000**
- What is your primary route path?: **/encounters**

You have successfully created your scaffolded Mediator! Now we can proceed. Import the project into your favourite IDE. You will see that Yeoman has created a maven project for you that contains the mediator code. In the folder **src/main/resources** are configuration files for the mediator. In addition, Yeoman will have scaffolded three source files:

- src/main/java/tutorial/MediatorMain.java
- src/main/java/tutorial/DefaultOrchestrator.java
- src/test/java/tutorial/DefaultOrchestratorTest.java

MediatorMain, like the name implies, contains the main entry point for the mediator. In addition it also loads the configuration. Here we will need to make a small edit in order to setup our routing from the HIM. In the **MediatorMain** class, there's a method called **buildRoutingTable()**. This is the configuration for the endpoints. Like the previous tutorial, we want to route to **/encounters/{anything}**, rather than just **/encounters**. Therefore, change the line

```
routingTable.addRoute("/encounters", DefaultOrchestrator.class);
```

to

```
routingTable.addRegexRoute("/encounters/.*", DefaultOrchestrator.class);
```

This tells the mediator engine that the **DefaultOrchestrator** class will handle any requests on the **/encounters/{anything}** endpoint. After this change we'll also need to ensure that the HIM Core will know how to route correctly to the mediator, so lets make sure all our config files have the correct values. Open up **src/main/resources/mediator.properties** and supply your OpenHIM config details and save:

```
mediator.name=Tutorial-Mediator
# you may need to change this to 0.0.0.0 if your mediator is on another server than HIM Core
mediator.host=localhost
mediator.port=4000
mediator.timeout=60000

core.host=localhost
core.api.port=8080
```

```
# update your user information if required
core.api.user=root@openhim.org
core.api.password=openhim-password
```

Open up **src/main/resources/mediator-registration-info.json** and update the details to match our new mediator path:

```
{
  ...
  "endpoints": [
    {
      ...
      "path": "..." //remove this
    }
  ],
  "defaultChannelConfig": [
    {
      "urlPattern": "/encounters/.*",
      ...
      "routes": [
        {
          ...
          "path": "..." //remove this
          ...
        }
      ]
    }
  ]
}
```

Next, take a look at **src/main/java/tutorial/DefaultOrchestrator.java**. This is the main landing point for requests from the HIM Core and this is the main starting point for writing your mediator code. The mediator engine uses the Akka framework, and **DefaultOrchestator** is an actor for processing requests. For now we will just setup a pass-through to the health record service, therefore you can edit the code as follows:

```
package tutorial;

import akka.actor.ActorSelection;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import org.openhim.mediator.engine.MediatorConfig;
import org.openhim.mediator.engine.messages.MediatorHTTPRequest;
import org.openhim.mediator.engine.messages.MediatorHTTPResponse;

import java.util.HashMap;
import java.util.Map;

public class DefaultOrchestrator extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    private final MediatorConfig config;

    private MediatorHTTPRequest originalRequest;

    public DefaultOrchestrator(MediatorConfig config) {
        this.config = config;
    }

    private void queryHealthRecordService(MediatorHTTPRequest request) {
        log.info("Querying the health record service");
        originalRequest = request;

        ActorSelection httpConnector = getContext().actorSelection(config.userPathFor("http-connector
        Map <string, string="">headers = new HashMap<>();
        headers.put("Accept", "application/json");

        MediatorHTTPRequest serviceRequest = new MediatorHTTPRequest(
                request.getRequestHandler(),
                getSelf(),
                "Health Record Service",
                "GET",
                "http",
                "localhost",
                3444,
                request.getPath(),
                null,
                headers,
                null
        );

        httpConnector.tell(serviceRequest, getSelf());
    }

    private void processHealthRecordServiceResponse(MediatorHTTPResponse response) {
        log.info("Received response from health record service");
        originalRequest.getRespondTo().tell(response.toFinishRequest(), getSelf());
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof MediatorHTTPRequest) {
            queryHealthRecordService((MediatorHTTPRequest) msg);
        } else if (msg instanceof MediatorHTTPResponse) {
            processHealthRecordServiceResponse((MediatorHTTPResponse) msg);
        } else {
            unhandled(msg);
        }
    }
}
```

When a request is received from core, the mediator engine will send a message to **onReceive**. When this happens we can trigger a request to the health record service, which we can do by referencing the **http-connector**. The http-connector is an actor provided by the engine for interacting with HTTP services. We look up this connector using an **ActorSelection** and send it an HTTP Request message, setting up the appropriate parameters for calling the health record service. When the connector receives a response from the health record service, it will respond to us by sending a HTTP Response message. Therefore in **onReceive**, we add handling for **MediatorHTTPResponse** and when receiving it we respond to the HIM Core with the health record. The original request (MediatorHTTPRequest) provides us with a handle for responding, and we can simply pass along the response message. Note that for orchestrator we don't need to worry about threading, blocking or anything like that: the Akka framework takes care of all of that! For this tutorial we'll just disable the unit test for the class (**src/test/java/tutorial/DefaultOrchestratorTest.java**), just add the **@Ignore** annotation:

```
...
@Test
@Ignore
public void testMediatorHTTPRequest() throws Exception {
...
```

Feel free to complete this test if you want to get the hang of writing these! (Tip: the **org.openhim.mediator.engine.testing.MockHTTPConnector** class can be used to setup a mock endpoint) Now we're ready to build and launch our mediator.

```
$ mvn install
$ java -jar target/tutorial-mediator-0.1.0-jar-with-dependencies.jar
```

SunCertPathBuilderException: unable to find valid certification path to requested target

If you are attempting to start your Java Mediator and you are experiencing a **SunCertPathBuilderException** error then you will need to follow the below mini tutorial to install the self signed certificate before you can continue. This mini tutorial is a short and quick version to get your self signed certificate installed. A more detailed and in-depth explanation can be found here. Lets start by first creating a new folder where we will install our self signed certificate.

```
$ mkdir installCert
$ cd installCert
```

Download the InstallCert.java.zip folder and extract the **InstallCert.java** script into our new **installCert** directory. We need to compile our **InstallCert.java** script to generate a **jssecacerts** file for us. Lets start this off by executing the below command which will generate two Java **.class** files for us:

```
$ javac InstallCert.java
```

Once you have compiled the **InstallCert.java** script we need to execute it by running the following command:

```
$ java InstallCert localhost:8080
```

Make sure that the port you supply is the same as the OpenHIM core API (**default is 8080**). The script will start executing and request that you **enter certificate to add to trusted keystore**. just reply with **1** and the script will continue executing. Once the script has completed successfully you should a message printed at the bottom reading **Added certificate to keystore 'jssecacerts' using alias 'localhost-1'** The **installCert** script has executed successfully and created a **jssecacerts** file for us which we need to copy to our **$JAVA_HOME\jre\lib\security** folder. Once you have placed the **jssecacerts** in the correct folder then you can start the mediator again and **SunCertPathBuilderException** error should no longer exist. **NB! You will need to restart your Mediator before the self signed certificate takes affect**

```
$ java -jar target/tutorial-mediator-0.1.0-jar-with-dependencies.jar
```

Navigate to the **Mediators** section on OpenHIM to see your Mediator'

3.3.3 Testing your shiny new mediator

Lets move onto the next part of this tutorial. You will notice that when our Mediator got registered it also created a channel for us. We will be using this Channel (**Tutorial Mediator**) You can delete the channel (**Tutorial Channel**) as it is no longer needed. Make sure that your services are running as explained in the pre-requisite section. Execute the CURL command to send a request to OpenHIM core and run through our Mediator

```
$ curl -k -u tut:tut https://localhost:5000/encounters/1
```

Internal Server Error

If you get an **Internal Server Error** then make sure that your service is running and that **Tutorial Mediator** is able to route to it. Update your channel route to use your inet addr instead of localhost

```
Example:  Channel -> route -> Host:  192.168.1.10
```

Your transaction should have routed through the mediator successfully and responded with the correct return object. You should notice a **Successful** record in your transactions on the OpenHIM console with the results. You can explore the transaction details page by clicking on the transaction in this list to see some of the details that we set in the mediator.

3.4 Message adaption using a mediator

This tutorial is a follow on from the previous **Creating a basic passthrough Mediator** tutorial. This tutorial assumes that you fully completed the previous mediator and got it to successfully route to the Health Record service and respond with the correct data.

3.4.1 NodeJS Mediator

In this tutorial we will be transforming the response we get from our Health Record service into a readable HTML format. This demonstrates the type of processing that a mediator could do, however it could also do much more. Depending on your project you may want to do a lot more processing into different formats or even enrich a message by making calls to external services to get extra information to include in the original message.

Let's get started, below the **Create Initial Orchestration** section in **app/index.js** we will be creating a variable that will hold our HTML converted response.

```
/* ############################ */
/* ##### HTML conversion  ##### */
/* ############################ */

/* ##### Construct Encounter HTML  ##### */
// first loop through all observations and build HTML rows
var observationsHtml = '';
for (i = 0; i < ctxObject.encounter.observations.length; i++) {
  observationsHtml += '    <tr>' + "\n" +
  '      <td>'+ctxObject.encounter.observations[i].obsType+'</td>' + "\n" +
  '      <td>'+ctxObject.encounter.observations[i].obsValue+'</td>' + "\n" +
  '      <td>'+ctxObject.encounter.observations[i].obsUnit+'</td>' + "\n" +
  '    </tr>' + "\n";
}

// setup the encounter HTML
var healthRecordHtml = '  <h3>Patient ID: #'+ctxObject.encounter.patientId+'</h3>' + "\n" +
'  <h3>Provider ID: #'+ctxObject.encounter.providerId+'</h3>' + "\n" +
'  <h3>Encounter Type: '+ctxObject.encounter.encounterType+'</h3>' + "\n" +
'  <h3>Encounter Date: '+ctxObject.encounter.encounterDate+'</h3>' + "\n" +
'  <table cellpadding="10" border="1" style="border: 1px solid #000; border-collapse: collapse">' + '
'    <tr>' + "\n" +
'      <td>Type:</td>' + "\n" +
'      <td>Value:</td>' + "\n" +
'      <td>Unit:</td>' + "\n" +
'    </tr>' + "\n" +
observationsHtml +
'  </table>' + "\n";

// setup the main response body
var responseBodyHtml = '<html>' + "\n" +
'<body>' + "\n" +
'  <h1>Health Record</h1>' + "\n" +
healthRecordHtml +
'</body>' + "\n" +
'</html>';
```

Once we have created our HTML response, we need to include it in the object we return. Replace **body: JSON.stringify(resp.body, null, 4),** with **body: responseBodyHtml,** in the response body and set the content type to **text/html**.

```
// update the response body with the 'responseBodyHtml' variable we created
var response = {
  ...
  headers: {
    'content-type': 'text/html'
  },
  body: responseBodyHtml,
  ...
};
```

Before we can test our updated mediator we first need to make sure we update our mediator version to indicate changes have been made. Open **package.json** and **app/config/mediator.json** and bump up the MINOR version by 1

```
{
  ...
  "version": "0.2.0",
  ...
}
```

Once we have changed our version number we can start our mediator again. Execute the following command to start up the server:

```
$ grunt serve
```

3.4.2 Java Mediator

In this tutorial we will be transforming the response we get from our Health Record service into a readable HTML format. From the last tutorial, we've already created a **processHealthRecordServiceResponse** method. Here we will be updating this method to convert the JSON response from the service into HTML. First we create a new class called **HealthRecord** to hold our health record model:

```
package tutorial;

public class HealthRecord {
    public static class Observation {
        private String obsType;
        private String obsValue;
        private String obsUnit;

        public String getObsType() {
            return obsType;
        }

        public void setObsType(String obsType) {
            this.obsType = obsType;
        }

        public String getObsValue() {
            return obsValue;
        }
        public void setObsValue(String obsValue) {
            this.obsValue = obsValue;
        }

        public String getObsUnit() {
            return obsUnit;
        }

        public void setObsUnit(String obsUnit) {
            this.obsUnit = obsUnit;
        }
    }

    private Integer patientId;
    private Integer providerId;
    private String encounterType;
    private String encounterDate;
    private Observation[] observations;

    public Integer getPatientId() {
        return patientId;
    }

    public void setPatientId(Integer patientId) {
        this.patientId = patientId;
    }

    public Integer getProviderId() {
        return providerId;
    }

    public void setProviderId(Integer providerId) {
        this.providerId = providerId;
    }

    public String getEncounterType() {
        return encounterType;
    }

    public void setEncounterType(String encounterType) {
        this.encounterType = encounterType;
    }

    public String getEncounterDate() {
        return encounterDate;
    }

    public void setEncounterDate(String encounterDate) {
        this.encounterDate = encounterDate;
    }

    public Observation[] getObservations() {
        return observations;
    }

    public void setObservations(Observation[] observations) {
        this.observations = observations;
    }
```

```
}
```

Next, in the **DefaultOrchestrator** class, create a method **parseHealthRecordJSON**:

```
private HealthRecord parseHealthRecordJSON(String healthRecord) {
    Gson gson = new GsonBuilder().create();
    return gson.fromJson(healthRecord, HealthRecord.class);
}
```

Now we'll be able to parse the health record response and use the model for HTML conversion. Let's set this up:

```
private String convertToHTML(HealthRecord healthRecord) {
    try {
        StringBuilder html = new StringBuilder("<html><body><h1>Health Record</h1>");
        html.append("<h3>Patient ID: #" + healthRecord.getPatientId() + "</h3>");
        html.append("<h3>Provider ID: #" + healthRecord.getProviderId() + "</h3>");
        html.append("<h3>Encounter Type: " + healthRecord.getEncounterType() + "</h3>");

        SimpleDateFormat from = new SimpleDateFormat("yyyymmdd");
        SimpleDateFormat to = new SimpleDateFormat("dd MMM yyyy");
        html.append("<h3>Encounter Date: " + to.format(from.parse(healthRecord.getEncounterDate())) +

        html.append("<table cellpadding=\"10\" border=\"1\" style=\"border: 1px solid #000; border-co
        html.append("<tr>" +"<td>Type:</td>" +"<td>Value:</td>" +"<td>Unit:</td>" +"</tr>");

        for (HealthRecord.Observation obs : healthRecord.getObservations()) {
            html.append("<tr><td>" + obs.getObsType() + "</td><td>" + obs.getObsValue() + "</td><td>"
        }

        html.append("</table></body></html>");
        return html.toString();
    } catch (ParseException ex) {
        originalRequest.getRequestHandler().tell(new ExceptError(ex), getSelf());
    }

    return null;
}

private void processHealthRecordServiceResponse(MediatorHTTPResponse response) {
    log.info("Received response from health record service");

    if (response.getStatusCode() == HttpStatus.SC_OK) {
        HealthRecord healthRecord = parseHealthRecordJSON(response.getBody());
        String html = convertToHTML(healthRecord);

        FinishRequest fr = new FinishRequest(html, "text/html", HttpStatus.SC_OK);
        originalRequest.getRespondTo().tell(fr, getSelf());
    } else {
        originalRequest.getRespondTo().tell(response.toFinishRequest(), getSelf());
    }
}
```

Lastly, let's bump up the version. In **pom.xml**

```
<version>0.2.0</version>
```

and in **src/main/resources/mediator-registration-info.json**

```
"version": "0.2.0",
```

You can now build and run your mediator as before:

```
$ mvn install
```

```
$ java -jar target/tutorial-mediator-0.2.0-jar-with-dependencies.jar
```

3.4.3 Testing your mediator

Now instead of using the CURL command, try using your web browser to test out a transaction: **https://localhost:5000/encounters/1**. You'll be prompted for login details - enter **tut** for both username and password. You may also need to instruct your browser to accept the self-signed certificate. If you have any issues doing so, you can also use the unsecure port instead: http://localhost:5001/encounters/1. You should now see the health record in your browser! The mediator has intercepted the request and done something useful with it.

You may create mediators for any additional processing that needs to occur for your project. Typical uses include **message transformation** (converting messages to a different format, either before they are send to another service or to convert the response from the other service as seen here) or for **message orchestration** (executing a business process for a message, eg. querying the client registry for an enterprise ID so that the message can be enriched with this information).

3.5 Orchestration with a mediator

In the previous tutorial we adapted the JSON response into HTML for easier viewing. In this tutorial we will learn how to enrich the response body with additional data by creating two orchestrations. This means we will be making a few requests to our tutorial services and joining them all together in the main response body.

This tutorial assumes that you have successfully completed the **Update Mediator for HTML Conversion** tutorial which we will be using to do our orchestrations.

3.5.1 NodeJS Mediator

For this tutorial we will be executing our Orchestrations asynchronously. For us to accomplish this we need to install another npm module called async. Run the below command to install and save it to our dependency list.

```
$ npm install async --save
```

We need to include our new module into our mediator so lets add it at the top of our script

```
...
var needle = require('needle');
var async = require('async');
```

Below the **Create Initial Orchestration** section we will be creating an array to store our orchestration requests

```
/* ######################################### */
/* ##### setup Orchestration Requests  ##### */
/* ######################################### */

// setup more orchestrations
orchestrations = [{
    ctxObjectRef: "client",
    name: "Get Client",
    domain: "http://localhost:3445",
    path: "/patient/"+resp.body.patientId,
    params: "",
    body: "",
    method: "GET",
    headers: ""
  }, {
    ctxObjectRef: "provider",
    name: "Get Provider",
    domain: "http://localhost:3446",
    path: "/providers/"+resp.body.providerId,
    params: "",
    body: "",
    method: "GET",
    headers: ""
  }];
```

Below the **Setup Orchestration Requests** section we will start our async code

```
/* ###################################### */
/* ##### setup Async Orch Requests  ##### */
/* ###################################### */

// start the async process to send requests
async.each(orchestrations, function(orchestration, callback) {

  // code to execute the orchestrations

}, function(err){

  // This section will execute once all requests have been completed
  // if any errors occurred during a request the print out the error and stop processing
  if (err){
    console.log(err)
    return;
  }

});
```

We will now have a look at the heart of the orchestrations. Inside the **async.each** replace **// code to execute the orchestrations** with the below code. This is the code that will send each orchestration request and push the orchestration data to the **orchestrationsResults** object

```
// construct the URL to request
var orchUrl = orchestration.domain + orchestration.path + orchestration.params;

// send the request to the orchestration
needle.get(orchUrl, function(err, resp) {

  // if error occured
  if ( err ){
    callback(err);
  }

  // add new orchestration to object
  orchestrationsResults.push({
    name: orchestration.name,
    request: {
      path : orchestration.path,
      headers: orchestration.headers,
      querystring: orchestration.params,
      body: orchestration.body,
      method: orchestration.method,
      timestamp: new Date().getTime()
    },
    response: {
      status: resp.statusCode,
      body: JSON.stringify(resp.body, null, 4),
      timestamp: new Date().getTime()
    }
  });

  // add orchestration response to context object and return callback
  ctxObject[orchestration.ctxObjectRef] = resp.body;
  callback();
});
```

We need to move our **HTML conversion** code and our **Construct Response Object** into our async process. We can place this directly after the check for any errors as this code should execute if no errors exist. Your async process should look like the below:

```
/* ###################################### */
/* ##### setup Async Orch Requests  ##### */
/* ###################################### */

// start the async process to send requests
async.each(orchestrations, function(orchestration, callback) {

  // construct the URL to request
  var orchUrl = orchestration.domain + orchestration.path + orchestration.params;

  // send the request to the orchestration
  needle.get(orchUrl, function(err, resp) {

    // if error occured
    if ( err ){
      callback(err);
    }

    // add new orchestration to object
    orchestrationsResults.push({
      name: orchestration.name,
      request: {
        path : orchestration.path,
        headers: orchestration.headers,
        querystring: orchestration.params,
        body: orchestration.body,
        method: orchestration.method,
        timestamp: new Date().getTime()
      },
      response: {
        status: resp.statusCode,
        body: JSON.stringify(resp.body, null, 4),
        timestamp: new Date().getTime()
      }
    });

    // add orchestration response to context object and return callback
    ctxObject[orchestration.ctxObjectRef] = resp.body;
    callback();
  });

}, function(err){

  // if any errors occured during a request the print out the error and stop processing
  if (err){
    console.log(err)
    return;
  }

  /* ############################ */
  /* ##### HTML conversion  ##### */
  /* ############################ */

  /* ##### Construct Encounter HTML  ##### */
  // first loop through all observations and build HTML rows
  var observationsHtml = '';
  for (i = 0; i < ctxObject.encounter.observations.length; i++) {
    observationsHtml += '    <tr>' + "\n" +
    '      <td>'+ctxObject.encounter.observations[i].obsType+'</td>' + "\n" +
    '      <td>'+ctxObject.encounter.observations[i].obsValue+'</td>' + "\n" +
    '      <td>'+ctxObject.encounter.observations[i].obsUnit+'</td>' + "\n" +
    '    </tr>' + "\n";
  }

  // setup the encounter HTML
  var healthRecordHtml = '  <h3>Patient ID: #'+ctxObject.encounter.patientId+'</h3>' + "\n" +
  '  <h3>Provider ID: #'+ctxObject.encounter.providerId+'</h3>' + "\n" +
  '  <h3>Encounter Type: '+ctxObject.encounter.encounterType+'</h3>' + "\n" +
  '  <h3>Encounter Date: '+ctxObject.encounter.encounterDate+'</h3>' + "\n" +
  '  <table cellpadding="10" border="1" style="border: 1px solid #000; border-collapse: collapse">'
  '    <tr>' + "\n" +
  '      <td>Type:</td>' + "\n" +
  '      <td>Value:</td>' + "\n" +
  '      <td>Unit:</td>' + "\n" +
  '    </tr>' + "\n" +
  observationsHtml +
  '  </table>' + "\n";

  // setup the main response body
  var responseBodyHtml = '<html>' + "\n" +
  '<body>' + "\n" +
  '  <h1>Health Record</h1>' + "\n" +
  healthRecordHtml +
  '</body>' + "\n" +
  '</html>';

  /* ###################################### */
  /* ##### Construct Response Object  ##### */
  /* ###################################### */

  var urn = mediatorConfig.urn;
  var status = 'Successful';
  var response = {
    status: 200,
    headers: {
      'content-type': 'application/json'
    },
    body: responseBodyHtml,
    timestamp: new Date().getTime()
  };

  // construct property data to be returned
  var properties = {};
  properties[ctxObject.encounter.observations[0].obsType] = ctxObject.encounter.observations[0].obsVa
  properties[ctxObject.encounter.observations[1].obsType] = ctxObject.encounter.observations[1].obsVa
  properties[ctxObject.encounter.observations[2].obsType] = ctxObject.encounter.observations[2].obsVa
  properties[ctxObject.encounter.observations[3].obsType] = ctxObject.encounter.observations[3].obsVa
  properties[ctxObject.encounter.observations[4].obsType] = ctxObject.encounter.observations[4].obsVa
  properties[ctxObject.encounter.observations[5].obsType] = ctxObject.encounter.observations[5].obsVa

  // construct returnObject to be returned
  var returnObject = {
    "x-mediator-urn": urn,
    "status": status,
    "response": response,
    "orchestrations": orchestrationsResults,
    "properties": properties
  }

  // set content type header so that OpenHIM knows how to handle the response
  res.set('Content-Type', 'application/json+openhim');
  res.send(returnObject);

});
```

We have a few more small additions to add before we have our Orchestration mediator complete. These steps are not crucial for the mediator to work but rather adds more value to the returned result. We will be updated the HTML that gets returned to include the patient details as well as the provider details that we retrieve in the orchestration calls that we make. Supply the below code underneath the **healthRecordHtml** variable.

```
/* ##### Construct patient HTML  ##### */
var patientRecordHtml = '  <h2>Patient Record: #'+ctxObject.client.patientId+'</h2>' + "\n" +
'  <table cellpadding="10" border="1" style="border: 1px solid #000; border-collapse: collapse">' + '
'    <tr>' + "\n" +
'      <td>Given Name:</td>' + "\n" +
'      <td>'+ctxObject.client.givenName+'</td>' + "\n" +
'    </tr>' + "\n" +
'    <tr>' + "\n" +
'      <td>Family Name:</td>' + "\n" +
'      <td>'+ctxObject.client.familyName+'</td>' + "\n" +
'    </tr>' + "\n" +
'    <tr>' + "\n" +
'      <td>Gender:</td>' + "\n" +
'      <td>'+ctxObject.client.gender+'</td>' + "\n" +
'    </tr>' + "\n" +
'    <tr>' + "\n" +
'      <td>Phone Number:</td>' + "\n" +
'      <td>'+ctxObject.client.phoneNumber+'</td>' + "\n" +
'    </tr>' + "\n" +
'  </table>' + "\n";



/* ##### Construct provider HTML  ##### */
var providerRecordHtml = '  <h2>Provider Record: #'+ctxObject.provider.providerId+'</h2>' + "\n" +
'  <table cellpadding="10" border="1" style="border: 1px solid #000; border-collapse: collapse">' + '
'    <tr>' + "\n" +
'      <td>Title:</td>' + "\n" +
'      <td>'+ctxObject.provider.title+'</td>' + "\n" +
'    </tr>' + "\n" +
'    <tr>' + "\n" +
'      <td>Given Name:</td>' + "\n" +
'      <td>'+ctxObject.provider.givenName+'</td>' + "\n" +
'    </tr>' + "\n" +
'    <tr>' + "\n" +
'      <td>Family Name:</td>' + "\n" +
'      <td>'+ctxObject.provider.familyName+'</td>' + "\n" +
'    </tr>' + "\n" +
'  </table>' + "\n";
```

We will also need to make sure that our new HTML variables gets added to our response body so lets add it to the **responseBodyHtml** variable.

```
// setup the main response body
var responseBodyHtml = '<html>' + "\n" +
'<body>' + "\n" +
'  <h1>Health Record</h1>' + "\n" +
healthRecordHtml +
patientRecordHtml +
providerRecordHtml +
'</body>' + "\n" +
'</html>';
```

One last thing we will be doing before we finish off our mediator is to add two new properties. These two properties will be constructed from the patient and provider object we got from our orchestrations. Add the two below properties.

```
var properties = {};
properties[ctxObject.client.givenName + ' ' + ctxObject.client.familyName + '(' + ctxObject.client.ge
properties[ctxObject.provider.title] = ctxObject.provider.givenName + ' ' + ctxObject.provider.family
...
```

Execute the following command to start up the server:

```
$ grunt serve
```

3.5.2 Java Mediator

We will enrich the health record service response using information from the client-service and the healthcare-worker-service. First we should setup object classes that can model the data, so let's create a new class **Patient**:

```
package tutorial;

public class Patient {
    private Integer patientId;
    private String familyName;
    private String givenName;
    private String gender;
    private String phoneNumber;

    public Integer getPatientId() {
        return patientId;
    }
    public void setPatientId(Integer patientId) {
        this.patientId = patientId;
    }

    public String getFamilyName() {
        return familyName;
    }

    public void setFamilyName(String familyName) {
        this.familyName = familyName;
    }

    public String getGivenName() {
        return givenName;
    }

    public void setGivenName(String givenName) {
        this.givenName = givenName;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}
```

and a new class **Provider**:

```
package tutorial;

public class Provider {
    private Integer providerId;
    private String familyName;
    private String givenName;
    private String title;

    public Integer getProviderId() {
        return providerId;
    }

    public void setProviderId(Integer providerId) {
        this.providerId = providerId;
    }

    public String getFamilyName() {
        return familyName;
    }
    public void setFamilyName(String familyName) {
        this.familyName = familyName;
    }

    public String getGivenName() {
        return givenName;
    }

    public void setGivenName(String givenName) {
        this.givenName = givenName;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

With these classes in place, we can look at orchestrating the requests. The flow we want to follow is

1. Query health record service
2. Lookup patient demographics for the patient with the id contained in the health record
3. Lookup healthcare demographics for the provider with the id contained in the health record
4. Convert the health record into HTML and insert the demographic information into this final response

Notice that both 2) and 3) can easily be separated from the health record orchestration, and in addition can easily run in parallel. Therefore, let's create separate actors for accomplishing these tasks. Let's create an actor for the task of resolving patients:

```
package tutorial;

import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import org.openhim.mediator.engine.messages.MediatorRequestMessage;
import org.openhim.mediator.engine.messages.SimpleMediatorRequest;
import org.openhim.mediator.engine.messages.SimpleMediatorResponse;

public class ResolvePatientActor extends UntypedActor {
    public static class ResolvePatientRequest extends SimpleMediatorRequest <integer>{
        public ResolvePatientRequest(ActorRef requestHandler, ActorRef respondTo, Integer requestObje
            super(requestHandler, respondTo, requestObject);
        }
    }

    public static class ResolvePatientResponse extends SimpleMediatorResponse <patient>{
        public ResolvePatientResponse(MediatorRequestMessage originalRequest, Patient responseObject)
            super(originalRequest, responseObject);
        }
    }

    LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private MediatorConfig config;
    public ResolvePatientActor(MediatorConfig config) {
        this.config = config;
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof ResolvePatientRequest) {
            //...
        } else {
            unhandled(msg);
        }
    }
}
```

We've defined an actor **ResolvePatientActor** and created two message types for it: **ResolvePatientRequest** and **ResolvePatientResponse**. So we expect a request message that'll ask the actor to resolve a patient. Let's add handling for this:

```
private ResolvePatientRequest originalRequest;

...

private void sendPatientRequest(ResolvePatientRequest request) {
    log.info("Querying the patient service");
    originalRequest = request;

    ActorSelection httpConnector = getContext().actorSelection(config.userPathFor("http-connector"));
    Map <string, string="">headers = new HashMap<>();
    headers.put("Content-Type", "application/json");

    String path = "/patient/" + request.getRequestObject();

    MediatorHTTPRequest serviceRequest = new MediatorHTTPRequest(
            request.getRequestHandler(),
            getSelf(),
            "Patient Service",
            "GET",
            "http",
            "localhost",
            3445,
            path,
            null,
            headers,
            null
    );

    httpConnector.tell(serviceRequest, getSelf());
}

@Override
public void onReceive(Object msg) throws Exception {
    if (msg instanceof ResolvePatientRequest) {
        sendPatientRequest((ResolvePatientRequest) msg);
    } else {
        unhandled(msg);
    }
}
```

When receiving a request, we will query the patient service for details matching the **requestObject**, here the patient id. Next we need to process the service response:

```
private Patient parsePatientJSON(String patient) {
    Gson gson = new GsonBuilder().create();
    return gson.fromJson(patient, Patient.class);
}

private void processPatientServiceResponse(MediatorHTTPResponse response) {
    Patient p = parsePatientJSON(response.getBody());
    ResolvePatientResponse actorResponse = new ResolvePatientResponse(originalRequest, p);
    originalRequest.getRespondTo().tell(actorResponse, getSelf());
}

@Override
public void onReceive(Object msg) throws Exception {
    if (msg instanceof ResolvePatientRequest) {
        sendPatientRequest((ResolvePatientRequest) msg);
    } else if (msg instanceof MediatorHTTPResponse) {
        processPatientServiceResponse((MediatorHTTPResponse) msg);
    } else {
        unhandled(msg);
    }
}
```

Next let's setup the analogous actor for resolving healthcare workers (or providers, to use another term):

```
package tutorial;

import akka.actor.ActorRef;
import akka.actor.ActorSelection;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import org.openhim.mediator.engine.MediatorConfig;
import org.openhim.mediator.engine.messages.*;

import java.util.HashMap;
import java.util.Map;

public class ResolveProviderActor extends UntypedActor {
    public static class ResolveProviderRequest extends SimpleMediatorRequest <integer>{
        public ResolveProviderRequest(ActorRef requestHandler, ActorRef respondTo, Integer requestObj
            super(requestHandler, respondTo, requestObject);
        }
    }

    public static class ResolveProviderResponse extends SimpleMediatorResponse <provider>{
        public ResolveProviderResponse(MediatorRequestMessage originalRequest, Provider responseObjec
            super(originalRequest, responseObject);
        }
    }

    LoggingAdapter log = Logging.getLogger(getContext().system(), this);
    private MediatorConfig config;
    private ResolveProviderRequest originalRequest;
    public ResolveProviderActor(MediatorConfig config) {
        this.config = config;
    }

    private void sendProviderRequest(ResolveProviderRequest request) {
        log.info("Querying the healthcare worker service");
        originalRequest = request;

        ActorSelection httpConnector = getContext().actorSelection(config.userPathFor("http-connector
        Map <string, string="">headers = new HashMap<>();
        headers.put("Content-Type", "application/json");

        String path = "/providers/" + request.getRequestObject();

        MediatorHTTPRequest serviceRequest = new MediatorHTTPRequest(
                request.getRequestHandler(),
                getSelf(),
                "Provider Service",
                "GET",
                "http",
                "localhost",
                3446,
                path,
                null,
                headers,
                null
        );

        httpConnector.tell(serviceRequest, getSelf());
    }

    private Provider parseProviderJSON(String provider) {
        Gson gson = new GsonBuilder().create();
        return gson.fromJson(provider, Provider.class);
    }

    private void processProviderServiceResponse(MediatorHTTPResponse response) {
        Provider p = parseProviderJSON(response.getBody());
        ResolveProviderResponse actorResponse = new ResolveProviderResponse(originalRequest, p);
        originalRequest.getRespondTo().tell(actorResponse, getSelf());
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof ResolveProviderRequest) {
            sendProviderRequest((ResolveProviderRequest) msg);
        } else if (msg instanceof MediatorHTTPResponse) {
            processProviderServiceResponse((MediatorHTTPResponse) msg);
        } else {
            unhandled(msg);
        }
    }
}
```

Now that we've got our actors, we can proceed with setting up the orchestrations in **DefaultOrchestrator**. Add two variables for keeping track of the orchestrations, as well as a variable for storing the parsed health record:

```
private HealthRecord healthRecord;
private Patient resolvedPatient;
private Provider resolvedProvider;
```

Next, after receiving the response from the health record service, instead of converting the record to HTML, we will use this point to create the resolve requests:

```
private void processHealthRecordServiceResponse(MediatorHTTPResponse response) {
    log.info("Received response from health record service");

    if (response.getStatusCode() == HttpStatus.SC_OK) {
        healthRecord = parseHealthRecordJSON(response.getBody());

        //Resolve patient
        ResolvePatientActor.ResolvePatientRequest patientRequest = new ResolvePatientActor.ResolvePat
                originalRequest.getRequestHandler(), getSelf(), healthRecord.getPatientId()
        );
        ActorRef patientResolver = getContext().actorOf(Props.create(ResolvePatientActor.class, confi
        patientResolver.tell(patientRequest, getSelf());

        //Resolve healthcare worker
        ResolveProviderActor.ResolveProviderRequest providerRequest = new ResolveProviderActor.Resolv
                originalRequest.getRequestHandler(), getSelf(), healthRecord.getProviderId()
        );
        ActorRef providerResolver = getContext().actorOf(Props.create(ResolveProviderActor.class, con
        providerResolver.tell(providerRequest, getSelf());
    } else {
        originalRequest.getRespondTo().tell(response.toFinishRequest(), getSelf());
    }
}
```

Then we just need to wait for the responses, and when completed we can format the final result and respond to the client:

```
private void finalizeRequest() {
    if (resolvedPatient==null || resolvedProvider==null) {
        //still waiting for results
        return;
    }

    String html = convertToHTML();
    FinishRequest fr = new FinishRequest(html, "text/html", HttpStatus.SC_OK);
    originalRequest.getRespondTo().tell(fr, getSelf());
}

@Override
public void onReceive(Object msg) throws Exception {
    if (msg instanceof MediatorHTTPRequest) {
        queryHealthRecordService((MediatorHTTPRequest) msg);
    } else if (msg instanceof MediatorHTTPResponse) {
        processHealthRecordServiceResponse((MediatorHTTPResponse) msg);
    } else if (msg instanceof ResolvePatientActor.ResolvePatientResponse) {
        resolvedPatient = ((ResolvePatientActor.ResolvePatientResponse) msg).getResponseObject();
        finalizeRequest();
    } else if (msg instanceof ResolveProviderActor.ResolveProviderResponse) {
        resolvedProvider = ((ResolveProviderActor.ResolveProviderResponse) msg).getResponseObject();
        finalizeRequest();
    } else {
        unhandled(msg);
    }
}
```

We've modified the **onReceive** method to wait for the response messages. When received we set the patient or provider objects and then check to see if the request can be finalized. As mentioned in a previous tutorial, we do not need to concern ourselves with threading or locks, as Akka will take care of all those aspects. Therefore we don't need to worry about synchronizing the **if (resolvedPatient==null || resolvedProvider==null)** line.

Lastly we just need to update the **convertToHTML** method:

```
private String convertToHTML() {
    try {
        StringBuilder html = new StringBuilder("<html><body><h1>Health Record</h1>");
        String patientName = resolvedPatient.getGivenName() + " " + resolvedPatient.getFamilyName();
        html.append("<h3>Patient Name: " + patientName + "</h3>");
        html.append("<h3>Patient Gender: " + resolvedPatient.getGender() + "</h3>");
        html.append("<h3>Patient Phone: " + resolvedPatient.getPhoneNumber() + "</h3>");

        String providerName = resolvedProvider.getTitle() + " " + resolvedProvider.getGivenName() + "
        html.append("<h3>Provider Name: " + providerName + "</h3>");

        html.append("<h3>Encounter Type: " + healthRecord.getEncounterType() + "</h3>");

        SimpleDateFormat from = new SimpleDateFormat("yyyymmdd");
        SimpleDateFormat to = new SimpleDateFormat("dd MMM yyyy");
        html.append("<h3>Encounter Date: " + to.format(from.parse(healthRecord.getEncounterDate())) +

        html.append("<table cellpadding=\"10\" border=\"1\" style=\"border: 1px solid #000; border-co
        html.append("<tr>" +"<td>Type:</td>" +"<td>Value:</td>" +"<td>Unit:</td>" +"</tr>");

        for (HealthRecord.Observation obs : healthRecord.getObservations()) {
            html.append("<tr><td>" + obs.getObsType() + "</td><td>" + obs.getObsValue() + "</td><td>"
        }

        html.append("</table></body></html>");
        return html.toString();
    } catch (ParseException ex) {
        originalRequest.getRequestHandler().tell(new ExceptError(ex), getSelf());
    }

    return null;
}
```

And that's it. Let's bump up the version to **0.3.0**, as per the previous tutorial, and then build and run the mediator as before:

```
$ mvn install
```

```
$ java -jar target/tutorial-mediator-0.3.0-jar-with-dependencies.jar
```

3.5.3 Testing your mediator

Try accessing the HIM using your web browser as per the previous tutorial: **https://localhost:5000/encounters/1**. You should not only see the health record, but also the patient and healthcare worker demographics! Also try looking at the transaction log in the HIM Console. You'll see that each orchestration is logged with the full request details with all the unformatted responses. We're all done :)

CHAPTER 4

User guide

4.1 Basic configuration

This getting started guide will take you through two very important parts of the OpenHIM console which will allow you to create **Clients** and **Channels** to get messages routed through the system.

Before you get started with **Clients** and **Channels** you will need OpenHIM core and OpenHIM console setup. To do so, follow the installation guide here.

To get a better understanding of what the openHIM core does and how it works, read up on the OpenHIM core concepts

A **Client** is usually some system that you want to able to send request to the OpenHIM. Setting up a **client** allows the OpenHIM to authenticate requests. A **Channel** defines a path that a request will take through the OpenHIM. It describes one more **routes** for the request to be forwarded to, which **clients** are allowed to use this **channel**, which requests are to be direccted to this **channel** and many more options that allow you to controls what happens for a particular request.

To manage **clients** and **channels** you will need to log into the OpenHIM console and then you may follow the steps below.

Note - Only an Admin user has the permission to Add/Edit/Delete a **Client** or **Channel**

4.1.1 Adding Clients

Follow the below steps to successfully create/update a **Client**

- Navigate to the **Clients** menu option found in the left sidebar.
- On the **Clients** page you will be presented with a list of all the created **Clients**
- Click on the blue "**+ Client**" button to open a popup modal box where you will supply the **Client** details **OR** click on one of the existing **Clients** to open up the popup modal with the **Clients'** saved details.
- Supply all the required fields (marked with a *) and click the blue "Save changes*" button when completed.

There are many fields that you may supply, here is an explanation of what each of them do:

- **Client ID** - This is a unique ID giving to a client to be used as a reference when adding **Channels** as well as for authorisation checking.
- **Client Name** - This is a descriptive name of the **Client**.
- **Domain** - A domain that is associated with this **Client** - **Note** The domain needs to match the CN of the certificate if a certificate is used otherwise the **Client** won't be authorised successfully.

- **Roles** - The **Client** roles field is a list of authorized user groups that are allowed to access this channel. You can either select a role from the suggested roles that appear when you start typing, or you can add a new role to the list by typing in the role and pressing "**Enter**"
- **Certificate** - The certificate field is used when the OpenHIM core is running using Mutual TLS Authentication and needs to authenticate requests coming from the **Client**. By default, OpenHIM core uses Mutual TLS Authentication
- **Basic Auth Password** - The password field is used when the OpenHIM core is running in basic auth mode and does not require a certificate, it does however require a password.

Note - Either a Certificate OR a Basic Auth Password is required depending on the configuration. If Basic Auth is enabled in the OpenHIM core configuration then only a password is required, if Mutual TLS Authentication is enabled then a **Client** Certificate is required.

Note - When a **Client** Certificate is added or updated, the OpenHIM console will inform the user that a server restart is required. This is for the new certificate to be applied correctly. The user can either decide to manually restart the server at a later time or to click the red "**Restart Server Now!**" button.

4.1.2 Adding Channels

Follow the below steps to successfully create/update a **Channel**

- Navigate to the **Channels** menu option found in the left sidebar.
- On the **Channels** page you will be presented with a list of all the created **Channels**
- Click on the blue "**+ Channel**" button to open a popup modal box where you will supply the **Channel** details **OR** click on one of the existing **Channels** to open up the popup modal with the **Channels'** saved details.
- Supply all the required fields and click the blue "**Save changes**" button when completed.

The two *most* important fields to supply are the **URL Pattern** field and the **Allowed roles and clients** field. The **URL Pattern** field describes which incoming requests should be send down this **channel**. It does this by looking at the URL of the incoming request and testing if it matches the RegEx that you supply in this field. Note, only the first matched **channel** that is found recieves the request for processing. The **Allowed roles and clients** field identifies which **clients** are allowed to send requests to this **channel**. If a request matches a **channel** but the **client** system is not specified in this field or a **role** containing that the **client** belongs to is not specified in this field then the request will be denied access to the **channel**.

There are many fields that you may supply and these are spread over a number of tabs, here is an explanation of what each of them do:

- **Basic Info tab**
  - **Channel Name** - This is a descriptive name of the **Channel**.
  - **Channel Type** - Select a **Channel** type
    - **HTTP** - Default **Channel** type.
    - **TCP** - Supply a TCP Host and Port
    - **TLS** - Supply a TLS Host and Port
    - **Polling** - Supply a Polling schedule - Cron format: '*/10 * * * *' or Written format: '10 minutes' - The module 'Agenda' is used to accomplish the polling - You can find more documentation here
  - **Status** - Set whether this channel is enabled to receive requests or if its disbaled*.
- **Request Matching tab**:

    - **URL Pattern** - Supply a URL pattern to match an incoming transaction - **Note** this field excepts a RegEx value - More information on RegEx can be found here or here
        - NB!. This field is not applicable for **Channel Type** of **TCP** or **TLS**
    - **Priority** - If a transaction matches the URL patterns of two or more channels, then the channel with higher priority will be picked. A value of 1 is the highest possible priority (first priority). Larger numbers therefore indicate that a channel should take lower priority.
    - **Authentication Type** - Set whether this channel is **Private** or **Public**
    - **Whitelisted IP addresses** - ???A list of IP addresses that will be given access without authentication required???
    - **Allowed roles and clients** - Only applicable when **Authentication Type** is set to **Private**. Supply the roles and **Clients** allowed to make requests to this channel
    - **Match Content Types** - Supply what content type to match too. (e.g text/json)
    - **Matching Options** - These options allows a **Channel** to be used if the request body matches certain conditions.
        - **No Matching** - No matching applicable
        - **RegEx Matching** - Supply a RegEx to match
        - **XML Matching** - Supply a X Path as well as a value to match
        - **JSON Matching** - Supply a JSON property as well as a value to match
- **Routes tab**:
    - **Mediator Route** - Select a mediator route if any, to populate the required route fields
    - **Name** - This is a descriptive name of the route
    - **Route Type** - Select whether this route is an HTTP/TCP or MLLP request
    - **Path** - Supply a path the route should follow. If none supplied then the **Channel** URL Pattern will be used.
    - **Path Transform** - Applies a said-like expression to the path string - Multiple endpoints can be reached using the same route.
    - **Host** - The host where this route should go to.
    - **Port** - The port where this route should go to.
    - **Basic Auth Username** - Supply a username if the route requires basic authentication.
    - **Basic Auth Password** - Supply a password if the route requires basic authentication.
    - **Is this the primary route?** - Set whether or not a route is primary - Setting a route to primary indicates that this is the first route to check and is the primary endpoint to reach.
    - **Status** - Set whether or not a route is enabled/disabled.
    - **+ Save** - All required fields need to be supplied before the blue "**+ Save**" button becomes active.
    - **Note** - At least one route needs to be added to the **Channel** and only one route is allowed to be set to primary
- **Data Control tab**:
    - **Store Request Body** - Select whether or not to store the request body.
        - **Note** - If a transaction is made through a POST/PUT/PATCH method and request body is NOT saved, then the transaction cannot be rerun.

   - **Store Response Body** - Select whether or not to store the response body.
   - **URL Rewriting enabled** - URL rewriting allows the OpenHIM to look for URLs in a response and rewrite them so that they point to the correct location.
     * **From Host/Port** - Supply the host and port value you are looking to rewrite.
     * **To Host/Port** - Supply the host and port value that will replace the 'From Host/Port' matches.
     * **Path Transform** - Applies a said-like expression to the path string - Multiple endpoints can be reached using the same route.
   - **Add Auto Rewrite Rules** - Determines whether automatic rewrite rules are used. These rules enabled URLs to be automatically rewritten for any URLs that points to a host that the OpenHIM proxies (any host on a primary route). These can be overridden by user specified rules if need be.
- **User Access tab**:
   - **User groups allowed to view this channel's transactions** - Supply the groups allowed to view this **Channel's** transactions
   - **User groups allowed to view this channel's transactions request/response body** - Supply the groups allowed to view the request/response body of this **Channel's** transactions
   - **User groups allowed to rerun this channel's transactions** - Supply the groups allowed to rerun this **Channel's** transactions
- **Alerts tab**:
   - **Status** - Supply the status of a transaction when the alert should be sent. This can be supplied in a range format (e.g 2xx or 4xx)
   - **Failure Rate (%)** - Supply the failure rate of when to start sending the alerts (e.g 50 - once failure rate above 50% then alerts will be sent)
   - **Add Users** - Add individual users to receive alerts
     * **User** - Select a user from the drop down to receive a alert
     * **Method** - Select the method of how the alert should be delivered [Email | SMS]
     * **Max Alerts** - Select the frequency of how often to send a alert [no max | 1 per hour | 1 per day]
   - **Add Groups** - Add an entire group to receive alerts
     * **Add a new group** - Select a group from the drop down to be added to alerts
   - **+ Alert** - All required fields need to be supplied before the blue "**+ Save**" button becomes active.

If you find a field that is not described here, please let us know by filing an issue on github with the 'documentation' label.

4.2 Adding Users

In order to configure the OpenHIM you have to be a registered user account with relevant permissions. A default super/admin user is provided when you first run the OpenHIM.

The default admin user is as follows:

```
username: root@openhim.org
password: openhim-password
```

Note: It is recommended that you change these as soon as you have installed the him to avoid abuse. Newer versions of the OpenHIM console should prompt you to do this on first login.

Using the default admin user, you may create other users. These too may belong to the admin group or may belong to other groups. Non-admin users cannot create clients and channels, however, they may view transactions for certain channels that they are given access to.

Note: Users that belong to the **admin** group are Super Users.

Users accounts are created in order to give users of the system an certain capabilities depending on the groups to which they belong. Users can access these capabilities through the OpenHIM console

4.2.1 How are users different from clients

Clients are different from users, they represent systems that can route transactions through the OpenHIM. Users are people accessing and configuring the OpenHIM and clients are systems that are allowed to send requests to the OpenHIM.

4.2.2 User Groups:

Groups are created automatically by just adding a new group name in the user form. You do not need to add a group explicitly. When you go on to create the channel, you just need to make sure the group name matches the one you specified when you created the user.

There are 2 kinds of group

1. The ‘admin’ group: this is a special group that grants users all permissions
2. Then the rest are defined by the system administrator and in the channels, an admin can set whether the group has any the permissions below.

4.2.3 Permissions

Users belonging to a certain group can be assigned certain permissions on a channel. This is done by adding the group to which they belong to that particular permission.

The permissions themselves are pretty self explanatory and are listed below with some brief explanations.

1. Can view channel transactions
2. Can view channel transaction bodies - bodies may contain private patient data
3. Can re-run transactions - enabling this permission needs to be done with care because it may cause downstream duplicates and data corruption if they users doesn’t know what they are doing

Also on the users page, there is a matrix that shows these permissions. This can be viewed by clicking the button above the list of users.

4.2.4 Walk through and examples

1. To add a user as an admin user, navigate to the admin section and click the button to add the user.

Required fields, are as follows:

1. Email - This needs to be a valid and unique email address
2. First Name

3. Last Name
4. Groups
5. Password and Confirm Password

Optional Fields are as follows:

1. MSISDN - the users cellphone number in the MSISDN format (eg. 27825555555) should you want to receive sms alerts
2. Receive daily reports, via email
3. Receive weekly reports, via email
4. Filter & List settings: here you may pre-define how you want to view your transactions

4.2.5 Reports

The two kinds of reports mentioned above send transaction metrics aggregated over a period. In these reports, you can see, the number of transactions that went through as well as their statuses.

The statuses are as follows:

1. Failed
2. Processing
3. Completed
4. Completed with errors
5. Successful

4.2.6 Filter and list settings

1. Filter settings: Here you set how you want to view transactions on the Transactions page by default. You can default it to show transactions by status by channel as well as limit the number of transactions per page.
2. List settings: Upon clicking on a transaction in the transactions page, you can choose by default whether to view the transaction on the same page, or to open it in a new window altogether.

If you find a field that is not described here, please let us know by filing an issue on github with the 'documentation' label.

4.3 Transaction List

The **transactions** page is pretty straight forward. It shows a list of the most recent transactions that the OpenHIM has recieved and processed. You are presented with a number of different filters (even more are accessible by clicking the 'Toggle Advanced Filters' button).

You may click on each transaction in the lsit to get more details on it. From here you can view the request and response details, re-run that transaction or view the different routes that it was sent to (if there are multiple).

If this transaction was routed though a medaitor you may see some additional details such as the orchestrations that the mediator performed.

Each transaction is marked with a status that show what state of processing it is in and whether the transaction was successful or not. Here is what each status means for a particular transaction:

- Processing - We are waiting for responses from one or more routes
- Successful - The primary route and all other routes returned successful http response codes (2xx).
- Failed - The primary route has returned a failed http response code (5xx)
- Completed - The primary route and the other routes did not return a failure http response code (5xx) but some weren't successful (2xx).
- Completed with error(s) - The primary route did not return a failure http response code (5xx) but one of the routes did.

4.3.1 Error resolution

If a transaction has failed or needs to be re-run you may do so by either clicking on the transaction and choosing 're-run transaction' or you can perform a bulk re-run by selecting the desired transactions and choosing 're-run selected transactions'. You may also choose to re-run all transactions that match a particular filter. Just filter by the desired parameters and click 're-run all transaction that match current filters'.

All bulk re-runs can be monitored in the 'tasks' page.

Note: the original transaction is always stored when a transaction is re-run, it is just marked as re-run. You will be warned if you try to re-run a transation that has already been re-run as this could cause duplicate data in your system.

4.4 Alerting and reports

The OpenHIM supports alerting users via email or sms under specific conditions. It is also able to send out daily and weekly reports about the transaction that it has processed. In the following section we explore these functions in more detail.

4.4.1 Failure alerting

Alerts can be sent out to a group of users when a particular http status code is recieved as a response to a transaction. To setup alerts, edit the channel that you wish to enable alerts for and select the 'Alerts' tab. On this tab you can add rules for when alerts are sent out. You must specify which http status code you want the alerts to fire on (eg. 401). You can even specify a range like 4xx for any status codes in the 400-499 range. You may also optionally set a failure rate. This allows you to only fire alerts if the rate of failure is above the percentage that you specify. Alerts are sampled at 1 min intervals.

To ensure that alerts are sent to the right group of people, you must specify the users or contact list that you want to recieve the alerts. You may choose individual users or choose to add an entire contact list of users. Contact list can be managed through the 'Contact lists' option on the main menu.

You may add as many alerts as you need and as many users and contact lists as you require for each alert.

4.4.2 Reports

The OpenHIM can also produce daily and weekly reports for users. These will contain information such as how many reuqest were processed and how many of those were successful or how many failed. There are two ways to setup reporting. I user may enable reporting on their profile (click on the username on the top right and choose profile, then enable the reports that you wish toi recieve) or an admin user can enable reporting for any other user. By default the dailt report are sent at 7am the following day and the weekly reports are sent out at 7am each Monday for the previous week.

4.5 Polling Channels (scheduled tasks)

A great feature of the OpenHIM is the ability to trigger tasks in other systems. This is made possible by a special type of channel called a polling channel. Polling channels are channels that the OpenHIM will trigger internally on some sort of schedule. When a channel is triggered it will cause each of the routes that are configured for that channel to execute.

The OpenHIM will trigger the polling channel with a `GET` request to the `/trigger` path on the defined schedule. Each route can override the path with their own as long as they are configured with a path. External systems can be triggered by pointing a route at them. The external systems will have to expose an HTTP endpoint for them to be triggered. The triggering will always happen as an HTTP `GET` request.

To configure a polling channel, in the console click on 'Channels' on the menu, choose add channel and set the type of the channel to 'polling'. You will be able to provide a schedule for the polling channel to be executed. You may provide this in cron format (ie. 0 4 * * *) or in a descriptive format (ie. 5 minutes). See the agenda documentation for a more complete description of this format. From there you may configure the rest of the channel as usual and add routes for each external system that is to be triggered.

4.6 Certificates

The OpenHIM has a built in capability to manage TLS certificates. You can upload a certificate and key that you have bought from a certificate authority such as Thwate or you can even generate your own self signed certificate to use in your private OpenHIM implementation. Both mechanisms are secure, however we suggest you purchase a certificate from a trusted certificate authority so save you some pain with self signed certificates.

The OpenHIM also allows you to trust particular certificates. This allows you to specify exactly which client or external hosts you trust and it ties in with the OpenHIMs authentication mechanism for clients.

4.6.1 Server certificate

To upload an OpenHIM server certificate simple drag and drop the certificate and key on the correct boxes on the certificates page. You will be asked to restart the OpenHIM for this to take effect. The OpenHIM will also warn you if the key and certificate pair that you have uploaded do not match. DO NOT restart the server if these don't match. It will prevent the server from being able to startup correctly and you will have to fix this manually in the database. If your key requires a passphrase be sure to submit that in the field provided as well.

Generating a server certificate

To generate a self signed certificate click on the '+ Create Server Certificate' button in the top right. This will guide you through the process of creating an certificate and key and it will automatically add this to the server once you are done. Make sure you download the certificate and key when asked to do so as the key is not stored on the server for security reasons.

4.6.2 Client certificates

If you have some client certificates or host certificates that you want the OpenHIM to trust you can add them by simply dropping them in the bottom box to upload them. These certificates may be attached to clients when you edit a particular client from the clients page and enable clients to be authenticated when using mutual TLS. They may also be used on a route when editing a channel to trust a particular hosts certificate.

Generating a trusted client certificate

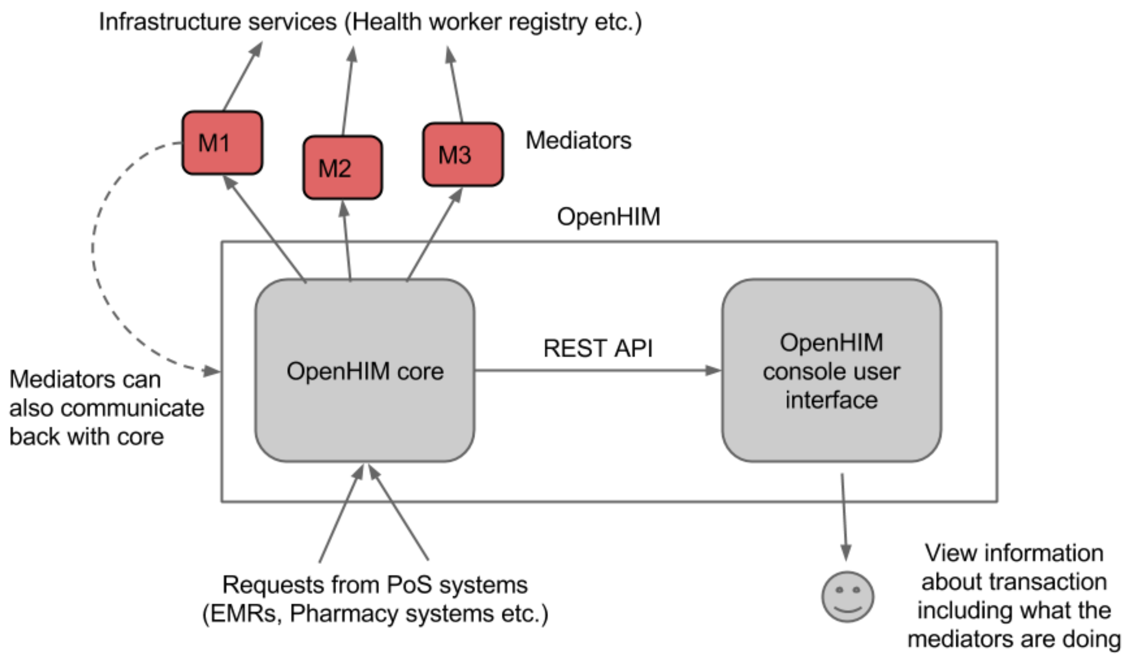
You may generate a client certificate by clicking the '+ Create Client Certificate' button and following the steps. Make sure you download the certificate and key when asked to do so as the key is not stored on the server for security reasons.

4.7 Mediators

OpenHIM mediators are separate micro services that run independently to the OpenHIM and perform additional mediation tasks for a particular use case. The common tasks within a mediator are as follows:

- Message format adaptation - this is the transformation of messages received in a certain format into another format (eg. HL7 v2 to HL7 v3 or MHD to XDS.b).
- Message orchestration - this is the execution of a business function that may need to call out to one or more other service endpoint on other systems. (eg. Enriching a message with a client's unique identifier retrieved from a client registry then sending the enriched message to a shared health record).

Mediators can be built using any platform that is desired (some good options are pure Java using our mediator engine, Node.js, Apache Camel, Mule ESB, or any language or platform that is a good fit for your needs). The only restriction is that the mediator MUST communicate with the OpenHIM-core in a particular way. Mediators must register themselves with the OpenHIM-core, accept request from the OpenHIM-core and return a specialised response to the OpenHIM-core to explain what that mediator did. A diagram of how mediators fit into the overall OpenHIM architecture can be seen below.


Infrastructure services (Health worker registry etc.)
M1
M2
M3
Mediators
OpenHIM
Mediators can also communicate back with core
OpenHIM core
REST API
OpenHIM console user interface
Requests from PoS systems (EMRs, Pharmacy systems etc.)
View information about transaction including what the mediators are doing

If you are interested in developing you own mediators see the documentation available here and see our tutorials page for specific examples to get you started.

4.7.1 Mediator Types

There are a few common types of mediators, these are described below.

Pass-through mediator

A Pass-through mediator just accepts a request and passes it on unchanged, these are not very useful and are only really used as a starting point for development.

Adaptor mediator

An Adaptor mediators accept a request and transform/adapt that request into another format before sending the request on to its final destination.

Orchestration mediator

An Orchestration mediator accepts a request and uses that request to execute some business process. This could involve making webservice calls to one or more other services to gather additional information about the request or to process it further. Finally a response is collated and returned to the OpenHIM.

4.7.2 Installing Mediators

Mediators may be developed in any language and only talk to the OpenHIM via its RESTful API. Therefore, the installation instructions will differ for each mediator. Please refer to the documentation of that mediator to details on how to install it. However, there are a few points that apply to all mediators:

- Mediators DO NOT have to be installed on the same server and the OpenHIM.
- You should ensure that the mediator is able to reach the OpenHIM-core servers RESTful API endpoint.
- You should ensure that the OpenHIM is able to reach the mediator's endpoint for recieving requests.
- You should ensure that you configure the mediator with correct credentials so that it may access the OpenHIMs RESTful API as an admin user.
- You should ensure that the mediator trust the OpenHIM-core's certificate (if it is self signed) as API communication MUST take place over https.

4.7.3 Existing Mediators

To find some existing mediators we suggest searching github for "openhim-mediator" which is the naming convension for OpenHIM mediators. For more information on writing you own mediator click here.

4.8 Sample disaster recovery procedure

This page describes a suggested disaster recover plan for an OpenHIM deployment. We suggest you follow this as a minimum for an production deploymnet of the OpenHIM.

4.8.1 Prior to disaster

The OpenHIM can be configured fairly simply to allow for disaster recovery. The only artefacts that need to be protected are the mongodb databases (main and the audit db) that the OpenHIM uses and the OpenHIM config file(s) that are used.

- For each mongo database you should use a geographically distributed replica set for redundancy. See here for more details. A 3 node set is suggested with 2 nodes in your primary data center and 1 node in a geographically distant location.
- The OpenHIM config file(s) should also be backed up. These should be periodically rsync'd to a geographically distant location (this can be on the same instance that your distant mongodb node is located).

A resource should also be created that describes the location of where the backup data is stored and contains the details of servers and procedure put in place through using this guide.

Security and firewalling

To secure the OpenHIM we suggest only allowing access to the specific port needed for the application to run. The following must be exposed, all others should be restricted.

- OpenHIM API port (default: 8080)
- OpenHIM non-secure transaction port (default: 5001)
- OpenHIM secure transaction port (default: 5000)
- Any TCP ports you have specified in the OpenHIM UI

We also suggest that the mongodb replica sets all be hosted on instances separate from the OpenHIM application with only the follow port allowed through the firewall:

- The mongodb port (default: 27017)

We also suggest that these instances block access from all other IPs other than the instance that the OpenHIM-core server is hosted on.

4.8.2 During a disaster

1. Ensure that a new primary mongodb node is elected if mongo was the failure
2. Ensure that the application remains operable.
3. Attempt to bring up the failed system. If this is a mongodb node ensure that is rejoins the replica set.

4.8.3 After the disaster

1. Ensure that no data was lost.
2. Identify root cause of the problem.
3. Attempt to mitigate the root cause of the problem.

4.8.4 Disaster Scenario Test

A schedule should be set up that outlines when the disaster recovery process is tested. Ideally the test scenario should be executed greater than zero times per project.

4.9 OpenHIM Core versioning and compatibility

The OpenHIM Core component uses Semantic Versioning. This means that if a specific software component, such as the OpenHIM Console or a Mediator states that it is compatible with Core version 1.2 for example, it means that:

- At a minimum the component is compatible with Core version 1.2 but is NOT guaranteed to work with a lower version of Core such 1.1
- The component WILL be compatible with any patch version in its release range, such as Core 1.2.1 or Core 1.2.2, even if the component was developed against a higher patch number such as 1.2.3
- WILL be compatible with Core 1.x, such as version 1.3 or 1.4, since these versions are backwards compatible with lower versions
- The software component is NOT guaranteed to work with Core 2.0 or higher, however this doesn't preclude the possibility that it CAN work.

4.10 Auditing

4.10.1 ATNA Audit Repository

The OpenHIM provides full support as an Audit Repository actor in the IHE ATNA profile.

You can make use of this functionality by enabling any of the audit servers in config before starting up the OpenHIM-core:

```
"auditing": {
  "servers": {
    "udp": {
      "enabled": true,
      "port": 5050
    },
    "tls": {
      "enabled": true,
      "port": 5051
    },
    "tcp": {
      "enabled": true,
      "port": 5052
    }
  },
  ...
}
```

The OpenHIM supports both RFC3881 and DICOM formatted audit events.

The OpenHIM-console has an audit viewer available on the 'Audit Log' page.

4.10.2 ATNA Audit Events

The OpenHIM will generate audit events on application start/stop, as well as user authentication. These events can either be sent to the OpenHIM's own internal audit repository, or to an external repository. This can be setup in config by choosing an appropriate `interface`:

```
"auditEvents": {
  "interface": "tls",
  "host": "192.168.1.11",
  "port": 8888
}
```

Options for the interface are: `internal`, `udp`, `tls` and `tcp`. The host and port does not need to be set for the `internal` interface.

Note that audit events are generated in RFC3881 format, but see our RFC3881 to DICOM Mediator for converting to DICOM.

CHAPTER 5

Developer guide

5.1 Design overview

This document will describe the architectural design of an Interoperability Layer for use within the OpenHIE project (The OpenHIM is a reference implementation of an OpenHIE Interoperability Layer). It describes the key components that should make up an interoperability layer and how this relates to the other services that are used in OpenHIE.

The interoperability layer considers of 2 separate sets of components:

1. The core (thin proxy) component
2. Orchestrators and adapter services

Together these 2 sets of components make up an interoperability layer. These can be seen in the diagram below. In the following section we will describe the requirement for the key requirements for the interoperability layer and explore each of these components in more details.

5.1.1 Key Requirements

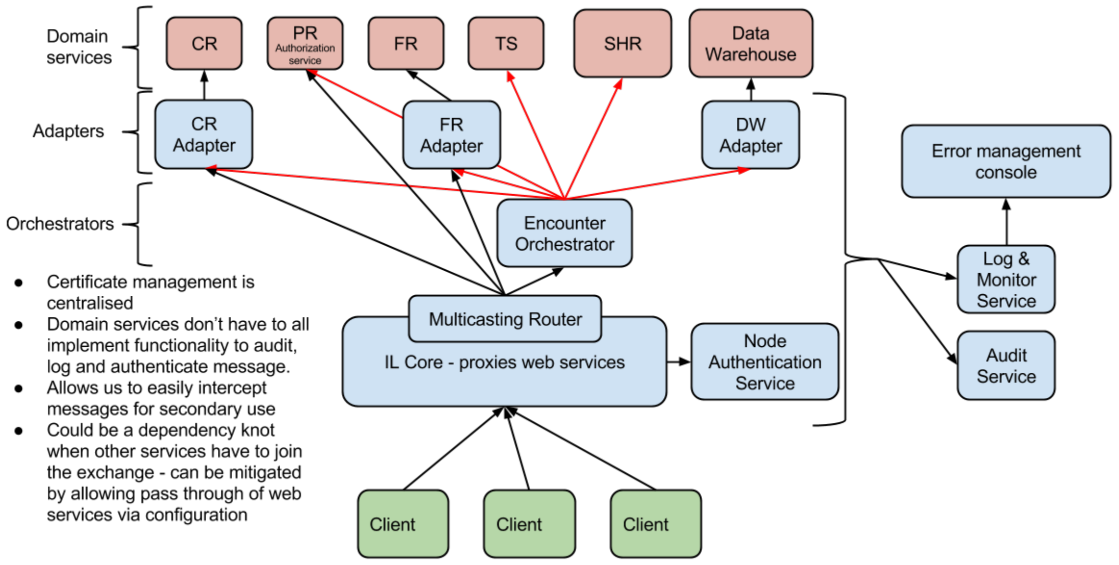
The full requirements that the Interoperability Layer community have identified can be found here: Interoperability Layer - Use Cases and Requirements

The following is a list of key requirements that are seen as necessary for an interoperability layer:

- Provide a central point for authentication and authorization to the HIE services.
- Log, audit and monitor communications for the components of the HIE.
- Provide error management and transaction monitoring information to the administrators of the HIE.
- Provide transaction orchestration services as well as adapter services to transform message between different message formats.

5.1.2 The architecture

Below, the architecture of the Interoperability Layer is shown in the context of the other expected services and registries. The Interoperability Layer components are show in **blue**, the domain services (registries) are shown in **red** and the clients are shown in **green.**


Domain services
Adapters
Orchestrators
CR
PR
Authorization service
FR
TS
SHR
Data Warehouse
CR Adapter
FR Adapter
DW Adapter
Error management console
Encounter Orchestrator
Log & Monitor Service
Audit Service
Multicasting Router
IL Core - proxies web services
Node Authentication Service
Client
Client
Client
Certificate management is centralised
Domain services don't have to all implement functionality to audit, log and authenticate message.
Allows us to easily intercept messages for secondary use
Could be a dependency knot when other services have to join the exchange - can be mitigated by allowing pass through of web services via configuration

The core (thin proxy) component

This component can be thought of as the entry point into the HIE. It provides some common mundane services so that other domain services don't have to implement these. This component basically just acts as a web service proxy while performing some additional functions on the incoming requests. The functions that this component should perform are as follow:

- Each message that is received from a client should be logged (stored in its entirely with metadata) and audited (store key information about that transaction and who ran it).
- Authentication and authorization services for the transaction within the HIE should be handled here.
- Displaying and monitoring errors that occur between the services, making use of the logging function to do this.
- A routing mechanism that routes requests received to the correct upstream service.

This component makes use of several other services in order to perform the functions mentioned above. These can be external services and we can likely use existing software components to fulfil these functions. The required services are explained below:

- Log service - This service stores each message in its entirety along with metadata about the message such as time and date the message was received and the response that the service returned.
- Audit service - This service audits each message received by storing an audit log entry. This log entry contains certain key information such as who sent the message, what information was requested and when the information was requested.
- Authorization and Authentication service - This service ensures that the person requesting or submitting information is known to the HIE and has the correct privileges to do so.

The interoperability layer core component contacts each one of these services when it receives a message to ensure the appropriate information is stored. It then passes the message on to the router where it is sent to the correct upstream service. The router makes use of a publish and subscribe pattern so that messages can be routed to multiple interested parties. This allows for secondary use of the messages received by the HIE. For example, encounter message could be routed to the SHR as well as an aggregation service where they could be aggregated and submitted to an aggregate data store such as a data warehouse.

The orchestrator and adapter services

This set of components provides services that manipulate the requests that are sent to them. They are often implementation specific so they will change as the use cases that the HIE supports change. Each of these components are separate, independent services that perform a specific function following the micro services architecture (click here for additional information about mico service architectures). There are 2 major types of these services:

1. Orchestrators - This service type enables a business process to be executed, this normally involves one or more additional services being invoked to process the required transaction.
2. Adapters – This service type adapts an incoming request to a form that the intended recipient of the request can understand.

These services are invoked whenever there is a need to orchestrate or adapt a certain transaction. If they are not needed the core interoperability layer component will just call the domain service directly. Orchestrators may use other adapters to send messages to other services. Designing these orchestrators and adapters as independent services allows for additional logic or business processes to be added to the HIE as the need arises. This allows the architecture to grow as the environment changes.

Adapters are used in 2 cases:

1. To simplify communication with the Domain services (for orchestrator use)
2. To adapt a standard-based interface to a custom domain service interface

Both the orchestrator and adapter services are also expected to log and audit messages that they send out to the domain services.

How authentication and authorization is handled within OpenHIE

The interoperability layer and system that it connects to will make use of the IHE ATNA profile's node authentication section for authentication. For authorization the provider registry will maintain a list of provider authorities and the interoperability layer will check these during orchestration of each transaction.

Derek Ritz has put together a great slideshow to show how authorization and authentication will be handled within OpenHIE. Please see this resource here: authentication and authorization slideshow.

5.1.3 Features of a central component

With the interoperability layer being a central component of the health information exchange there are a number of features that become apparent. Some of these are positive features and other are negative features. These are listed below:

- Certificate management is centralised, this allows for easier management and setup.
- Domain services don't have to all implement functionality to audit, log and authenticate message thus making them simpler.
- Allows messages to be easily intercepted for secondary use which is beneficial to enable additional functions as the HIE grows.
- Could be a dependency knot when other services have to join the exchange as this central component will have to be configured for each change - could be mitigated by allowing simple pass through of web services via configuration, thus the changes are in configuration only.

5.2 Detailed design using Node.js

Note: this design document was written before the development OpenHIM an as such some of the detail have changed or evolved with the OpenHIM's continuted development. It is a good starting point but not a complete picture.

Node.js is a good technology option on which to develop the interoperability layer core component for the following reasons:

- It is very lightweight
- It has a robust HTTP library
- It has robust support from 3rd party libraries for reading and modifying HTTP requests
- It is highly performant

5.2.1 Libraries to use

- Koa - Koa is a new web application framework for node.js. It provides easy mechanisms to expose web endpoints and process requests and responses in a stack-like approach.
- Passport.js - Middleware to provide authentication mechanisms (in the current implementation this has not yet been used).

5.2.2 General Overview

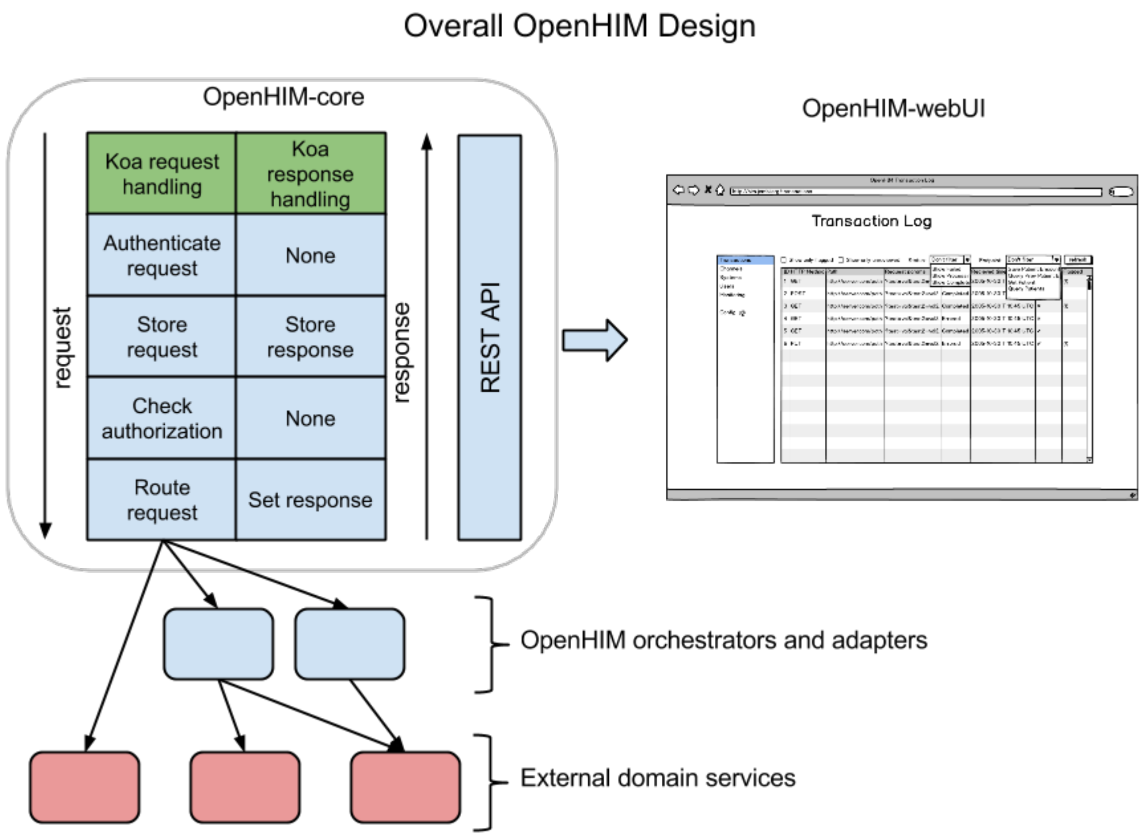
The Koa framework provides an easy way to modify HTTP request and responses as they are being processed. Each step that the OpenHIM needs to perform can be written as Koa middleware. Each middleware can handle a different aspect of processing that the OpenHIM need to perform such as authentication, authorization, message persistence and message routing. Developing each of these steps as Koa middleware allows them to be easily reused and allows us to add new steps for future versions. Koa stack approach to processing requests also fit our usecase well as it allows the middleware to affect both the request and the response as it is travelling through the system.

The Koa framework also gives us some convenience ctx.request and ctx.respose objects that are designed to be used for web applications but they are equally useful for handling web services.

5.2.3 Design

The OpenHIM-core will use Koa middleware to act on HTTP requests and Responses. Koa allows you to setup a stack of middleware, each middleware is called in order and gets an opportunity to do something with the request (going down the stack) and is then suspended. Once the end of the stack is reached Koa traverses back up the stack allowing each middelware to do something with the response.

Each row in the diagram representing the OepnHIM-core is a middleware component. Each of the components of the OpenHIM-core will be described further in the following sections. The OpenHIM-core will also have a REST API that will allow a web UI to be created for easy of management.



5.2.4 Authentication and Authorization

The are two possible combinations of authentication that the interoperability layer should provide to determine a client's identity:

- HTTP basic authentication
- ATNAs Node Authentication (PKI)

Once identify has been established the IL core component should check if that client has the authority to access the requested service.

The HIM should also provide a single-sign-on (SSO) facility to allow users of the HIE management application to have a single identity that they may used to access these applications. To do this the HIM should also act as an openid provider and provide functions to manage SSO users.

The two main workflows that we wish to enable for authentication and authorization are described in the following workflows:

- Common message security workflow
- SSO User workflow

Authentication

Client details for authentication are stored in the MongoDB database is the following format. Either a password or a certificate (in binary form) is stored in this structure depending on whether the user chooses to use PKI or HTTP basic auth to authenticate clients.

The OpenHIM application should allow new clients to be added and managed with the following details:

```
{
    "clientID": "Musha_OpenMRS",
    "domain": "him.jembi.org",
    "name": "OpenMRS Musha instance",
    "roles": [ "OpenMRS_PoC", "PoC" ],
    "passwordHash": "",
    "cert": ""
}
```

Basic auth

When authentication is set to HTTP basic auth then Koa middleware is setup to intercept the request as soon as it enters the HIM as shown above. This middleware will read client details (username and password-hash) out of the MongoDB store to determine if the client can be authenticated. If the client is rejected an error is returned else the request is considered authenticated and is then passed onto the authorization step.

PKI - mutual TLS

When the authentication method is set to PKI then the node http server must be setup to use https and it must be set to trust only clients that have a certificate that it knows of (is stored in a client's details). The domain of a client (identified in its certificate) will be used to map a request received from a client to its details as stored by the OpenHIM (shown above).

To help perform the authentication the passport.js module will be use. This provides us with middleware for a number of different authentication schemes. There is also Koa middleware available for passport.

Authorization

The OpenHIM only performs simple authorisation based on the path that is being requested. It should be able to restrict access to certain paths to clients with particular roles. Roles are identified in each client's details. The channel description shown in the router section below shows that each path has one or more allowed roles or clients associated with it. The authorisation component will check if the authenticated client has the authority to access the current path. If authorized the request will be passed on, else, the request will be denied and a HTTP 401 message will be returned.

5.2.5 Message persistence

Each request and response will be persisted so that it can be logged and so that error'd transaction may be re-run. This persistence occurs at two stages. Firstly, once a request is authenticated and authorised and secondly once a response has been received from the external service. All the metadata about a transaction is stored in a single document in MongoDB. The relevant sections are just updated as new information is received. The structure of this information is shown below.

In addition the ability to store orchestration steps exists in the structure. We anticipate exposing a web service to enable mediators to report requests and responses that they make to/receive from external services and have these stored alongside the actual transaction.

```
{
    "_id": "123",
    "status": "Processing|Failed|Completed",
    "clientID": "Musha_OpenMRS",
    "request": {
        "path": "/api/test",
        "headers": {
            "header1": "value1",
            "header2": "value2"
        },
        "querystring": "param1=value1&param2=value2",
        "body": "<HTTP body>",
        "method": "POST",
        "timestamp": "<ISO 8601>"
    },
    "response": {
        "status": 201,
        "body": "<HTTP body>",
        "headers": {
            "header1": "value1",
            "header2": "value2"
        },
        "timestamp": "<ISO 8601>"
    },
    "routes": [
        {
            "name": "<route name>"
            // Same structure as above
            "request": { ... },
            "response": { ... }
        }
    ]
    "orchestrations": [
        {
            "name": "<orchestration name>"
            // Same structure as above
            "request": { ... },
            "response": { ... }
        }
    ]
    "properties": { // optional meta data about a transaction
        "prop1": "value1",
        "prop2": "value2"
    }
}
```

5.2.6 Router

The router allows request to be forwarded to one or more external services (these could be mediators or an actual HIE component). It does this by allowing the user to configure a number of channels. Each channel matches a certain path and contains a number of routes on which to forward requests. Request may be forwarded to multiple routes however there can only be one **primary route**. The primary route is a the route whose response is returned back to the service requester making use of the OpenHIM.

Channel will be able to be added, removed and updated dynamically as the application is running.

Channel may be access controlled via the 'allow' field. This field will specify a list of users or groups that are allowed to send requests to that channel. If a channel receives a request from an un-authorised source it will return an error.

A custom router will have to be developed that can route according to these rules. The router can be build using the node.js functions provides to make HTTP request and responses can be relayed using the .pipe() function.

```
[
    {
        "name": "Some Registry Channel",
        "urlPattern": "test/sample/.+",
        "allow": "*",
        "routes": [
            {
                "name": "Some Registry",
                "path": "some/other/path" // this is optional if left out original path is used
                "host": "localhost",
                "port": 8080
            }

        ],
        "properties": [ // optional meta data about a channel
            { "prop1": "value1" },
            { "prop2": "value2" }
        ]
    },
    {
        "name": "Some Registry Channel",
        "urlPattern": "test/sample2/.+/test2",
        "allow": [ "Alice","Bob", "PoC" ],
        "routes": [
            {
                "name": "Some Registry",
                "host": "localhost",
                "port": 8080,
                "primary": true
            },
            {
                "name": "Logger",
                "host": "log-host",
                "port": 4789
            }
        ]
        "properties": [ // optional meta data about a channel
            { "prop1": "value1" },
            { "prop2": "value2" }
        ]
    }
]
```

5.2.7 Restful API

The OpenHIM must also expose a restful API that enables it to be configured and to allow access to the transaction that it has logged. This restful API will drive a web application that can allow the OpenHIM to be configured and will allow allow transaction to be viewed and monitored.

The API must supply CRUD access to the following constructs:

- transaction logs
- transaction channels

- client details

It should also allow for the following actions:

- single and batch re-processing of transactions
- querying for monitoring statistics

The API reference as it current exist can be found here.

API Authentication

For details follow the following issue: https://github.com/jembi/openhim-core-js/issues/57#issuecomment-44835273

The users collection should look as follows:

```
{
    "firstname": "Ryan",
    "surname": "Crichton",
    "email": "r..@jembi.org",
    "username": "ryan.crichton",
    "passwordHash": "xxxxx",
    "passwordSalt": "xxxxx",
    "apiKey": "fd41f5da-b059-45e8-afc3-99896ee5a7a4",
    "groups": [ "Admin", "RHIE"]
}
```

5.3 Getting started with OpenHIM development

The fist thing you will need to do is get you development environment up. This guide describes how to get a development environment up for the OpenHIM-core and the OpenHIM-console.

5.3.1 Setting up your OpenHIM-core dev environment

You can use vagrant if you would want to get up and running quickly with a dev environment in a vm. See here to use Vagrant to fire up an instance. Otherwise, read on to learn more.

Clone the `https://github.com/jembi/openhim-core-js.git` repository.

Ensure you have the following installed:

- Node.js 0.12.0 or greater
- MongoDB (in Ubuntu run `sudo apt-get install mongodb`, in OSX using Homebrew, run `brew update` followed by `brew install mongodb`)

The OpenHIM core makes use of the Koa framework, which requires node version 0.12.0 or greater. Node also has to be run with the `--harmony` flag for Koa to work as it needs generator support.

The easiest way to use the latest version of node is to install `nvm`. On Ubuntu, you can install using the install script but you have to add `[[ -s $HOME/.nvm/nvm.sh ]] && . $HOME/.nvm/nvm.sh # This loads NVM` to the end of your `~/.bashrc` file as well.

Once `nvm`is installed, run the following:

```
nvm install 0.12
```

```
nvm alias default 0.12
```

The latest version of node 0.12 should now be installed and set as default. The next step is to get all the required dependencies using `npm`. Navigate to the directory where the openhim-core-js source is located and run the following:

```
npm install
```

Then build the project:

```
grunt build
```

In order to run the OpenHIM core server, MongoDB must be installed and running.

To run the server, execute:

`npm start` (this runs `grunt build` then `node --harmony lib/server.js` behind the scenes)

The server will by default start in development mode using the mongodb database 'openhim-development'. To start the serve in production mode use the following:

```
NODE_ENV=production node --harmony lib/server.js
```

This starts the server with production defaults, including the use of the production mongodb database called 'openhim'.

This project uses mocha as a unit testing framework with should.js for assertions and sinon.js for spies and mocks. The tests can be run using `npm test`.

Pro tips:

- `grunt watch` - will automatically build the project on any changes.
- `grunt lint` - ensure the code is lint free, this is also run before an `npm test`
- `npm link` - will symlink you local working directory to the globally installed openhim-core module. Use this so you can use the global openhim-core binary to run your current work in progress. Also, if you build any local changes the server will automatically restart.
- `grunt test --mochaGrep=<regex>` - will only run tests with names matching the regex
- `grunt test --ddebugTests` - enabled the node debugger while running unit tests. Add `debugger` statements and use `node debug localhost:5858` to connect to the debugger instance.

5.3.2 Setting up your OpenHIM-console dev environment

Clone the repository at `https://github.com/jembi/openhim-console.git` and then run `npm install`

Install cli tools: `npm install -g grunt-cli grunt bower`

Install bower web components: `bower install`

To run the unit tests run `grunt test`

To start up a development instance of the webapp run `grunt serve`. The hostname and port can be changed in `Gruntfile.js`. The hostname can be changed to `0.0.0.0` in order to access the site from outside.

Note all changes will be automatically applied to the webapp and the page will be reloaded after each change. In addition JSHint will be run to provide information about errors or bad code style. The unit tests will also be automatically be run if JSHint does not find any problems.

For unit testing we are using mocha with chai.js for assertions. We are using the BDD `should` style for chai as it more closely resembles the unit testing style that is being used for the OpenHIM-core component

This code was scaffolded using Yeoman and the angular generator. You can find more detials about the commands available by looking at the docs of those tools.

5.4 Developing mediators

OpenHIM mediators are separate micro services that run independently from the OpenHIM-core and perform additional mediation tasks for a particular use case. The common tasks within a mediator are as follows:

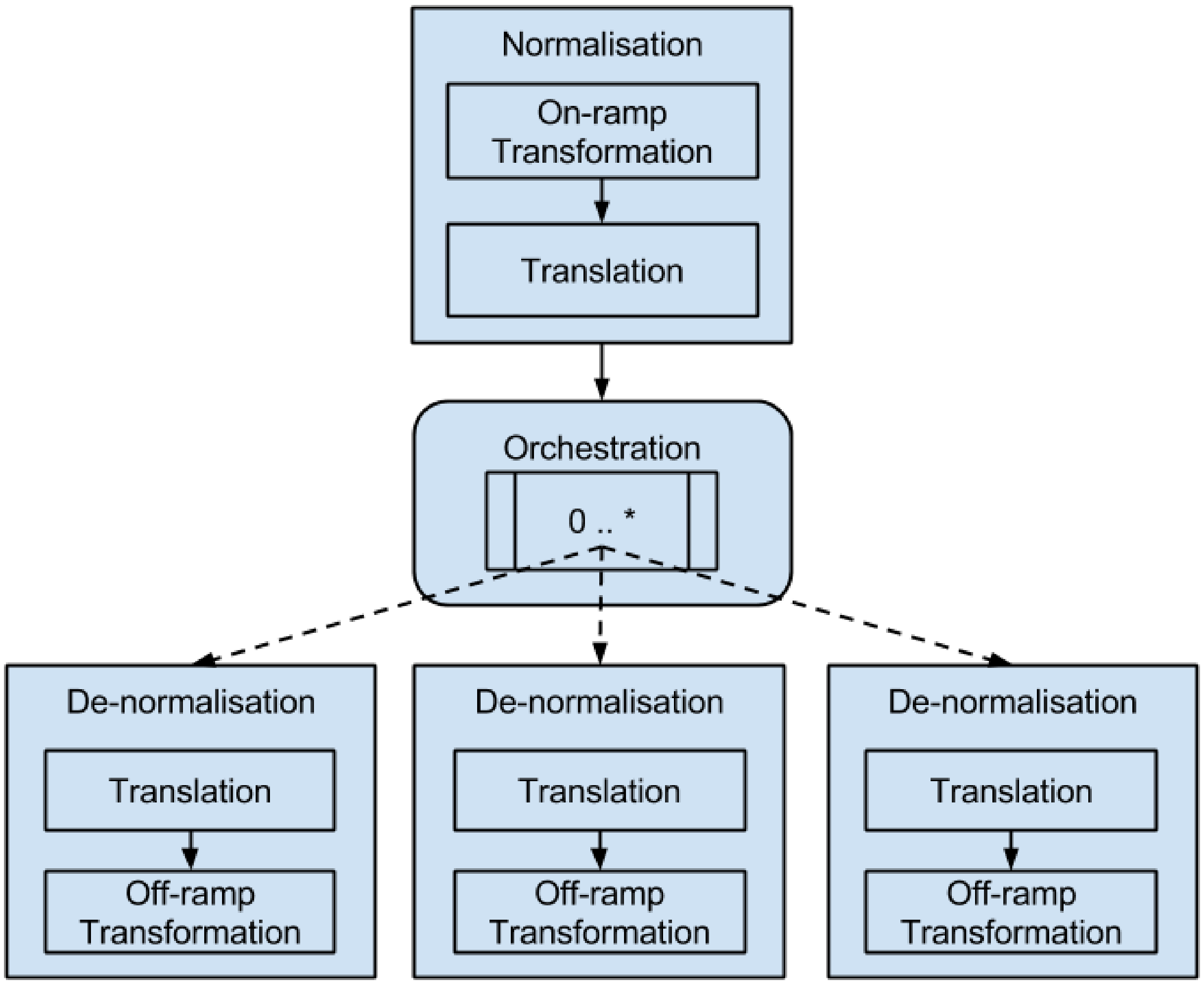
- Message format adaptation - this is the transformation of messages received in a certain format into another format (e.g. HL7 v2 to HL7 v3 or MHD to XDS.b).
- Message orchestration - this is the execution of a business function that may need to call out to other service endpoints on other systems. (e.g. Enriching a message with a client's unique identifier retrieved from a client registry).

Mediators can be built using any platform that is desired (some good options are Apache Camel, Mule ESB, or any language or platform that is a good fit for your needs). The only restriction is that the mediator MUST communicate with the OpenHIM-core in a particular way. There are 3 different types of communication that a mediator can have with the OpenHIM-core. These are described later.

You can also take a look at our handy mediator yeoman generators to get set-up with scaffolding to start building a mediator. To help you get started we have also created some tutorials that you can find here. If you're a java developer, you can also take a look at our mediator engine for additional documentation.

5.4.1 Suggested mediator structure

For maximum reusability and modifiability, we suggest that mediators be split into a number of sub-components. These sub-components are shown in the diagram bellow. Mediators do not need to follow this structure however it provides some useful benefits. If a mediator is simple and does not need the complexity added by having multiple sub-components it may implement its functionality in which ever way is simplest. If you mediator does not require this, you may skip this section.



Each mediator should consist of a **normalisation** sub-components, a **orchestration** sub-component and a **de-normalisation** sub-component. The purpose of each of these are described below.

Note: These description are taken the published thesis of Ryan Crichton: 'The Open Health Information Mediator: an Architecture for Enabling Interoperability in Low to Middle Income Countries'

Normalisation sub-component

This sub-component transforms the request message contained within a transaction to a normalised state. This normalised state is called the canonical form for that transaction. After this process the transaction data must be in a consistent and predictable format to allow components following this step to process it in a predictable fashion, no matter what format it arrived in. This process consists of two operations. Firstly, an on-ramp transformation is applied. This ensures that the message is transformed into a form that the HIM can process, thus enabling syntactic interoperability for the transaction. For example, if the transaction arrives from a legacy application that only supported exporting data in a custom XML format, this process would ensure that the XML is transformed into the canonical form that the HIM can understand, such as an HL7 version 2 message. Secondly, a translation operation is invoked. This operation is responsible for ensuring the codes and code systems used within the transaction are translated to a standard set of vocabulary or clinical terms, called reference terms, that have a common interpretation by other components of the HIM. This could involve a call to a terminology service to translate and verify that the codes used within the transaction are represented in, or are translated to, known reference terms. In this way semantic interoperability between service requesters and providers is achieved.

Orchestration sub-component

This sub-component is responsible for performing implementation-specific orchestration for the current transaction. The aim of the orchestration component is to execute the received transaction and perform any consequent action(s) required for this transaction. This could include zero or more calls to external services as well as the execution of business logic. This component compiles the response for the executed transaction and returns this to the persistence component which forwards the response to the service requester via the interface component. The calls to external systems should be done in parallel where possible to ensure that the orchestration is done quickly and efficiently as possible.

De-normalisation sub-component

This sub-component is responsible for transforming or constructing a service request in a format that is understandable to the service provider. This operates in a similar way to the normalisation component except the operations occur in the reverse order. This approach serves to decouple service providers from the orchestration component, which allows for service providers to be easily modified or replaced with minimal impact on the mediation component.

Separating the mediator into these difference components allows the same orchestration logic to be reused with multiple inbound and outbound message formats. It also allows the normalisation and de-normalisation sub-components to be split out of the mediator and scaled and load balanced independently from it. This is especially useful in high load applications. We recommend that mediation platform such as Mule ESB or Apache Camel be used to ease the construction of such a mediator simpler.

5.4.2 Mediator communication with core

Mediator registration

A mediator **MUST** register itself with the OpenHIM-core each time it starts up. The registration process informs the OpenHIM-core of some useful details:

- An identifier and name to associate with the OpenHIM-core
- The hostname or IP address of the mediator
- Default channel configuration that should be applied to the OpenHIM-core that the mediator needs
- The endpoints that the mediator exposes that the OpenHIM can contact it on.

In order to register itself a mediator MUST send an API request to the OpenHIM-core with the following format:

`POST https://<openhim-core_host>:<api_port>/mediators`

with a JSON body that conforms to the following structure:

```
{
    "urn": "<a unique URN>", // A unique identifier to identify the mediator, this identifier should
    "version": "", // the version of the mediator, if this is incremented the OpenHIM-core will updat
    "name": "", // a human readable name for the mediator
    "defaultChannelConfig": [ // (optional) an array of default channels to add for this mediator
        { ... }, // a channel object as defined by the OpenHIM-core - see https://github.com/jembi/op
        { ... }
    ],
    "endpoints": [ // (A minimum of 1 endpoint must be defined) an array of endpoints that the mediat
        { ... }, // a route object as defined by OpenHIM-core - see https://github.com/jembi/openhim-
        { ... }
    ],
    "configDefs": [ ... ], // (optional) An array of config definitions of config that can be set in
    "config": { "<param1>": "<val1>", "<param2>": "<val2>" } // (optional) Default mediator configura
}
```

The `configDefs` property defines an array of configuration definitions that each describe configuration parameters that could be provided by the user. These configuration parameters could have the following `type` properties:

- `string` - A string of text
- `bigstring` - A string of text that is expected to be large (it will be displayed as a text area on the OpenHIM-console)
- `bool` - A boolean value (true or false)
- `number` - An integer or decimal value
- `option` - A value from a pre-defined list. If this datatype is use then the `values` property MUST also be used. The `values` property specifies an array of possible values for the parameter.
- `map` - Key/value pairs. A map is formatted as an object with string values, e.g. `{ "key1": "value1", "key2": "value2" }`. New key/value pairs can be added dynamically.
- `struct` - A collection of fields that can be of any of type. If a parameter is a struct, then a `template` field MUST be defined. A template is an array with each element defining the individual fields that the struct is made up of. The definition schema is the same as the `configDefs` schema with the exception that a struct may not recursively define other structs.

A config definition may also specify an `array` property (boolean). If true, then the config can have an array of values. The elements in the array must be of the specified type, e.g. if the config definition is of type `string`, then the config must be an array of strings.

The OpenHIM-core SHALL respond with a HTTP status of 201 if the mediator registration was successful. The OpenHIM-core SHALL respond with an appropriate 4xx status if the mediator registration could not be completed due to a bad request. The OpenHIM-core SHALL respond with an appropriate 5xx status if the mediator registration could not be completed due to server error in the OpenHIM-core.

Mediator Config Definition Examples

Basic Settings The following is a config definition for basic server settings:

```
{
  ...
  "configDefs": [
    {
      "param": "host",
      "displayName": "Host",
      "description": "Server host",
      "type": "string"
    }, {
      "param": "port",
      "displayName": "Port",
      "description": "Server port",
      "type": "number"
    }, {
      "param": "scheme",
      "displayName": "scheme",
      "description": "Server Scheme",
      "type": "option",
      "values": ["http", "https"]
    }
  ]
}
```

Valid config would be:

```
{
  "host": "localhost",
  "port": 8080,
  "scheme": "http"
}
```

Map example A map is a collection of key/value pairs:

```
{
  ...
  "configDefs": [
    {
      "param": "uidMappings",
      "displayName": "UID Mappings",
      "type": "map"
    }
  ]
}
```

Valid config would be:

```
{
  "uidMappings": {
    "value1": "a1b2c3",
    "value2": "d4e5f6",
    "value3": "g7h8i9"
  }
}
```

Note that the keys `value1`, `value2`, etc. were not predefined in the definition. The OpenHIM-console allows users to dynamically add key/value pairs for a map.

Struct example A struct is a grouping of other types:

```
{
  ...
  "configDefs": [
    {
      "param": "server",
      "displayName": "Target Server",
      "description": "Target Server",
      "type": "struct",
      "template": [
        {
          "param": "host",
          "displayName": "Host",
          "description": "Server host",
          "type": "string"
        }, {
          "param": "port",
          "displayName": "Port",
          "description": "Server port",
          "type": "number"
        }, {
          "param": "scheme",
          "displayName": "scheme",
          "description": "Server Scheme",
          "type": "option",
          "values": ["http", "https"]
        }
      ]
    }
  ]
}
```

Valid config would be:

```
{
  "server": {
    "host": "localhost",
    "port": 8080,
    "scheme": "http"
  }
}
```

Array example The following is a config definition for a string array:

```
{
  ...
  "configDefs": [
    {
      "param": "balancerHosts",
      "displayName": "Balancer Hostnames",
      "description": "A list of hosts to load balance between",
      "type": "string",
      "array": true
    }
  ]
}
```

Valid config would be:

```
{
  "balancerHosts": [
    "192.168.0.1",
    "192.168.0.3",
    "192.168.0.7"
  ]
}
```

Arrays are supported for all types, including structs:

```
{
  ...
  "configDefs": [
    {
      "param": "balancerHosts",
      "displayName": "Balancer Hostnames",
      "description": "A list of hosts to load balance between",
      "type": "struct",
      "array": true,
      "template": [
        {
          "param": "host",
          "type": "string"
        }, {
          "param": "weight",
          "type": "number"
        }
      ]
    }
  ]
}
```

Valid config would be:

```
{
  "balancerHosts": [
    {
      "host": "192.168.0.1",
      "weight": 0.6
    }, {
      "host": "192.168.0.3",
      "weight": 0.2
    }, {
      "host": "192.168.0.7",
      "weight": 0.2
    }
  ]
}
```

Return transaction metadata

A mediator **SHOULD** return a structured object that indicates the response that should be returned to the user as well as metadata about the actions that were performed. The mediator is not required to do this however useful information can be returned to the OpenHIM-core in this way. If a structured response is not returned to the OpenHIM-core then what ever is returned to the OpenHIM-core is pass directly on to the client that make the request.

The structured object should be returned in the HTTP response for each request that the OpenHIM-core forwards to the mediator. If the mediator chooses to return a strucutred response then the mediator MUST return this object with a content-type header with the value: 'application/json+openhim'. If the mediator wants to set a specific content-type to return to the client, they can set this in the response object as a header (see below).

The JSON object returned to the OpenHIM should take the following form:

```
{
    "x-mediator-urn": "<a unique URN>", //same as the mediator's urn
    "status": "Successful", // (optional) an indicator of the status of the transaction, this can be
    "response": { ... }, // a response object as defined by OpenHIM-core - see https://github.com/jem
    "orchestrations": [ // (optional) an array of orchestration objects
        { ... }, // orchestration object as defined by OpenHIM-core - see https://github.com/jembi/op
        { ... }
    ],
    "properties": { // (optional) a map of properties that the mediator may want to report
        "pro1": "val",
        "pro2": "val"
    }
}
```

(Optional) Send heartbeats and recieve user configuration directly from OpenHIM-core

A mediator **MAY** opt to send heartbeats to the OpenHIM-core to demonstrate its aliveness. The heartbeats also allow it to recieve user specified configuration data and any changes to that configuration in a near real-time fashion.

The mediator can do this by utilising the mediator heartbeats API endpoint of the OpenHIM-core. You can find details on this endpoint here. This API endpoint, if supported by the medaitor, should always be called once at mediator startup using the `config:  true` flag to get the initial startup config for the mediator if it exists. There after the API endpoint should be hit at least every 30s (a good number to work with is every 10s) by the mediator to provide the OpenHIM-core with its heartbeat and so that the medaitor can recieve the latest user config as it becomes available.

(not yet implemented) Return transaction metrics

In addition to returning transaction metadata, a mediator MAY return transaction metrics about the transaction that is processes. To do this then a mediator MAY add a metrics object to the structured response object. This metrics object should be populated with any metrics that the mediator wishes to report.

The OpenHIM-core must be be setup to use a metrics service for this function to be used. The metrics object MUST be formatted as follows (see https://github.com/jembi/openhim-core-js/issues/104 for more details):

```
"metrics": {
    "<metric_name>": "62", // for metrics that apply to the entire transaction
    "<orchestration_name>.<metric_name>": "16", // for metrics that apply to a particular orchestrat
    ...
}
```

5.5 RESTful API

Each and every API call that is made to the OpenHIM has to be authenticated. The authentication mechanism that is used can be fairly complex to work with however it provides decent security.

The authentication mechanism is based on http://stackoverflow.com/a/9387289/588776.

5.5.1 Initial authentication notification

The user notifies the API that it wants to use its authenticated service:

`GET https://<server>:8080/authenticate/<user_email>`

If you don't have a user account yet, you can use the root user. The default root user details are as follows:

username: root password: openhim-password (you should change this on a production installation!)

The server will respond with the salt that was used to calculate the clients passwordHash (during user registration):

```
{
    "salt": "xxx",
    "ts": "xxx"
}
```

You must calculate a passwordhash using the received salt and the supplied user password. `passwordhash = (sha512(salt + password))`

5.5.2 For subsequent requests to the API

For every request you must add the following additional HTTP headers to the request:

```
auth-username: <username>
auth-ts: <the current timestamp - in the following format '2014-10-20T13:19:32.380Z' - user time must
auth-salt: <random uuid salt that you generate>
auth-token: <= sha512(passwordhash + auth-salt + auth-ts) >
```

The server will authorise this request by calculating sha512(passwordhash + auth-salt + auth-ts) using the password-hash from its own database and ensuring that:

- this is equal to auth-token
- the auth-ts isn't more than 2 seconds old

If these 2 conditions true the request is allowed.

5.5.3 Example implementations

An example of how this authentication mechanism can implemented for use with curl is show here: https://github.com/jembi/openhim-core-js/blob/master/resources/openhim-api-curl.sh

An example of how this is implemented in the OpenHIM Console see: https://github.com/jembi/openhim-console/blob/master/app/scripts/services/login.js#L12-L39 and https://github.com/jembi/openhim-console/blob/master/app/scripts/services/authinterceptor.js#L20-L50

5.5.4 API Reference

Channels resource

Channels represent configuration setting of how to route requests through the OpenHIM.

`https://<server>:<api_port>/channels`

Fetch all channels

`GET /channels`

The response status code will be `200` if successful and the response body will contain an array of channel objects. See the channel schema.

Add a channel

`POST /channels`

with a json body representing the channel to be added. See the channel schema.

The response code will be `201` is successful.

Fetch a specific channel

`GET /channels/:channelId`

where `:channelId` is the `_id` property of the channel to fetch.

The response status code will be `200` if successful and the response body will contain a channel object. See the channel schema.

Update a channel

`PUT /channels/:channelId`

where `:channelId` is the `_id` property of the channel to update and with a json body representing the channel updates. See the channel schema.

The response code will be `200` if successful.

Delete a channel

`DELETE /channels/:channelId`

where `:channelId` is the `_id` property of the channel to delete.

The response code will be `200` if successful.

Manually Trigger Polling Channel

'POST /channels/:channelId/trigger'

where ':channelId' is the '_id' property of the channel to manually trigger.

Clients resource

Other system that send request for the OpenHIM to forward.

`https://<server>:<api_port>/clients`

Fetch all clients

`GET /clients`

The response status code will be `200` if successful and the response body will contain an array of client objects. See the clients schema.

Add a client

`POST /clients`

with a json body representing the client to be added. See the clients schema.

The response code will be `201` is successful.

Fetch a specific client

`GET /clients/:clientId`

where `:clientId` is the `_id` property of the client to fetch.

The response status code will be `200` if successful and the response body will contain a client object. See the client schema.

Fetch a specific client by domain

`GET /clients/domain/:clientDomain`

where `:clientDomain` is the `domain` property of the client to fetch.

The response status code will be `200` if successful and the response body will contain a client object. See the client schema.

Update a client

`PUT /clients/:clientId`

where `:clientId` is the `_id` property of the client to update.

The response code will be `200` if successful.

Delete a client

`DELETE /clients/:clientId`

where `:clientId` is the `_id` property of the client to delete.

The response code will be `200` if successful.

Roles resource

Allows for the management of client access control to channels.

It should be noted that there is no actual roles collection in the database. The API is a facade on top of the `allow` and `roles` fields from Channels and Clients respectively. Roles can therefore also be altered by changing values for those fields directly.

Fetch all roles

`GET /roles`

The response status will be `200` if successful and the response body will contain an array of role objects, each consisting of a `name`, an array of `channels` and an array of `clients`, e.g.:

```
[
  {
    "name": "Role1",
    "channels": [
      {
        "_id": "56d56f34131d779a3f220d6d",
        "name": "channel1"
      },
      {
        "_id": "56dfff5ef51fbdc660fe6722",
        "name": "channel2"
      }
    ],
    "clients": [
      {
        "_id": "56d43e584582beae226d8226",
        "clientID": "jembi"
      }
    ]
  },
  {
    "name": "Role2",
    "channels": [
      {
        "_id": "56d43e424582beae226d8224",
        "name": "Channel3"
      }
    ],
    "clients": [
      {
        "_id": "56d43e584582beae226d8226",
        "clientID": "jembi"
      }
    ]
  },
  {
    "name": "internal",
    "channels": [
      {
        "_id": "56e116d9beabfb406e0e7c91",
        "name": "Daily Task"
      }
    ],
    "clients": []
  }
]
```

Fetch a specific role by role name

`GET /roles/:name`

The response status code will be `200` if successful and the response body will contain a role object in the same format as the role elements in the *Fetch all roles* operation response above. E.g.

```
{
  "name": "Role1",
  "channels": [
    {
      "_id": "56d56f34131d779a3f220d6d",
      "name": "channel1"
    },
    {
      "_id": "56dfff5ef51fbdc660fe6722",
      "name": "channel2"
    }
  ],
  "clients": [
    {
      "_id": "56d43e584582beae226d8226",
      "clientID": "jembi"
    }
  ]
}
```

Add a new role

`POST /roles`

with a json body containing the role name and channels and clients to apply to. At least one channel or client has to be specified. Channels and clients can be specified either by their `_id` or `name` for a channel and `clientID` for a client.

An example role that will give a client named *jembi* permission to access *channel1* and *channel2*.

```
{
  "name": "Role1",
  "channels": [
    {
      "name": "channel1"
    },
    {
      "name": "channel2"
    }
  ],
  "clients": [
    {
      "clientID": "jembi"
    }
  ]
}
```

The response status code will be `201` if successful.

Update an existing role

`PUT /roles/:name`

with a json body containing any updates to channels and clients. As with the *Add a new role* operation, channels and clients can be specified either by their `_id` or `name` for a channel and `clientID` for a client.

Note that the channel and client arrays, if specified, must contain the complete list of items to apply to, i.e. roles will be removed if they exist on any channels and clients that are not contained in the respective arrays. This also means that if `channels` and `clients` are specified as empty arrays, the result will be the same as deleting the role. If the fields are not specified, then the existing setup will be left as is.

The following example will change `Role1` by giving the clients *jembi* and *client-service* permission to access *channel1*. Any other channels will be removed, e.g. following from the *Add a new role* example above, access to *channel2* will be removed:

```
{
  "channels": [
    {
      "name": "channel1"
    }
  ],
  "clients": [
    {
      "clientID": "jembi"
    },
    {
      "clientID": "client-service"
    }
  ]
}
```

Roles can also be renamed by specifying the `name` field.

The response status code will be `200` if successful.

Delete an existing role

`DELETE /roles/:name`

Remove an existing role from all channels and clients.

The response status code will be `200` if successful.

Users resource

Console and API Users of the system.

`https://<server>:<api_port>/users`

Fetch all users

`GET /users`

The response status code will be `200` if successful and the response body will contain an array of users objects. See the user schema.

Add a user

`POST /users`

with a json body representing the user to be added. See the users schema.

The response code will be `201` is successful.

Fetch a specific user by email address

`GET /users/:email`

where `:email` is the `email` property of the user to fetch.

The response status code will be `200` if successful and the response body will contain a user object. See the user schema.

Update a user

`PUT /users/:email`

where `:email` is the `email` property of the user to update and with a json body representing the user updates. See the user schema.

The response code will be `200` if successful.

Delete a user

`DELETE /users/:email`

where `:email` is the `email` property of the user to delete.

The response code will be `200` if successful.

Transactions resource

Transactions store details about request and responses send through specifc channels.

`https://<server>:<api_port>/transactions`

An important concept to grasp with a transaction is the meaning of a transactions status. Here is a description of what each state means:

- Processing - We are waiting for responses from one or more routes
- Successful - The primary route and all other routes returned successful http response codes (2xx).
- Failed - The primary route has returned a failed http response code (5xx)
- Completed - The primary route and the other routes did not return a failure http response code (5xx) but some weren't successful (2xx).
- Completed with error(s) - The primary route did not return a failure http response code (5xx) but one of the routes did.

Fetch all transactions

`GET /transactions`

The response status code will be `200` if successful and the response body will contain an array of transaction objects. See the transaction schema.

The following query parameters are supported:

- `filterLimit`: The max number of transactions to return
- `filterPage`: The page to return (used in conjunction with `filterLimit`)
- `filterRepresentation`: Determines how much information for a transaction to return; options are
  - `simple`: minimal transaction information
  - `simpledetails`: minimal transaction information, but with more fields than simple

  - `bulkrerun`: minimal transaction information required in order to determine rerun status
  - `full`: Full transaction information
- `channelID`: Only return transactions that are linked to the specified channel
- `filters`: Advanced filters specified as an object. Transaction fields can be specified based on the transaction schema. For example, in order to filter by response status 200 and a property called `prop` with a value `val`, the following query could be used: `/transactions?filterLimit=100&filterPage=0&filters=%7B%22response.status%22:%22200%22,`

Add a transaction

`POST /transactions`

with a json body representing the transaction to be added. See the transactions schema.

The response code will be `201` is successful.

Fetch a specific transaction

`GET /transactions/:transactionId`

where `:transactionId` is the `_id` property of the user to fetch.

The response status code will be `200` if successful and the response body will contain a transaction object. See the transaction schema.

Find transactions by client Id

`GET /transactions/clients/:clientId`

where `:clientId` is the `clientID` property of the client we wish to find transaction for.

The response status code will be `200` if successful and the response body will contain an array of transaction objects. See the transaction schema.

Update a transaction

`PUT /transactions/:transactionId`

where `:transactionId` is the `_id` property of the transaction to update and with a json body representing the transaction updates. See the transaction schema.

The response code will be `200` if successful.

Delete a transaction

`DELETE /transactions/:transactionId`

where `:transactionId` is the `_id` property of the transaction to delete.

The response code will be `200` if successful.

###Contact groups resource

A contact group (or contact list) defines logical groups of users used for contacting users en masse.

`https://<server>:<api_port>/groups`

Fetch all groups

`GET /groups`

The response status code will be `200` if successful and the response body will contain an array of group objects. See the contact groups schema.

Add a group

`POST /groups`

with a json body representing the group to be added. See the contact groups schema.

The response code will be `201` is successful.

Fetch a specific group

`GET /groups/:groupId`

where `:groupId` is the `_id` property of the group to fetch.

The response status code will be `200` if successful and the response body will contain a group object. See the contact group schema.

Update a group

`PUT /groups/:groupId`

where `:groupId` is the `_id` property of the group to update.

The response code will be `200` if successful.

Delete a group

`DELETE /groups/:groupId`

where `:groupId` is the `_id` property of the group to delete.

The response code will be `200` if successful.

Tasks resource

Tasks are used to submit transactions to be re-run.

`https://<server>:<api_port>/tasks`

Fetch all tasks

`GET /tasks`

The response status code will be `200` if successful and the response body will contain an array of task objects. See the tasks schema.

Add a task

```
POST /tasks
```

with a json body representing the task to be added in the following format:

```
{
  "tids": [
    "id#1",
    "id#2",
    ...
    "id#N"
  ],
  "batchSize": 4,    //optional
  "paused": true     //optional
}
```

The `tids` field should contain an array of transaction identifiers indicating the transactions that need to be rerun. The `batchSize` field indicates the number of transactions that the core should run concurrently. To prevent a task from immediately starting upon add, the `paused` field can be added. In this case the task will simply be scheduled with a `Paused` status, ready to be started at any later stage.

The response code will be `201` if successful.

Fetch a specific task

```
GET /tasks/:taskId
```

where `:taskId` is the `_id` property of the task to fetch.

The response status code will be `200` if successful and the response body will contain a task object. See the task schema.

Update a task

```
PUT /tasks/:taskId
```

where `:taskId` is the `_id` property of the task to update.

Tasks can be paused, resumed or cancelled by sending through an update with status equal to `Paused`, `Queued` or `Cancelled` respectively.

The response code will be `200` if successful.

Delete a task

```
DELETE /tasks/:taskId
```

where `:taskId` is the `_id` property of the task to delete.

The response code will be `200` if successful.

Mediators

```
https://<server>:<api_port>/mediators
```

Fetch all mediators

`GET /mediators`

The response status code will be `200` if successful and the response body will contain an array of mediator objects. See the mediators schema.

Add a mediator

`POST /mediators`

with a json body representing the mediator to be added. See the mediators schema.

The response code will be `201` is successful.

Fetch a specific mediator

`GET /mediators/:urn`

where `:urn` is the `urn` property of the mediator to fetch.

The response status code will be `200` if successful and the response body will contain a mediator object. See the mediator schema.

Mediator heartbeat endpoint

This endpoint allows a mediator to send a heartbeat to the OpenHIM-core. This serves two purposes:

1. It allows the mediator to demonstrate its alive-ness and submit an uptime property
2. It allows the mediator to fetch the latest configuration from the OpenHIM-core

This endpoint only returns mediator config if the config has changed between the time the previous heartbeat was received and the current time. You may force the endpoint to return the latest config via the `config: true` property.

`POST /mediators/:urn/heartbeat`

where `:urn` is the `urn` property of the mediator that is sending in a heartbeat.

with an http body of:

```
{
  "uptime": 50.25 // The uptime is seconds
  "config": true // (Optional) a flag to force the OpenHIM-core to send back the latest config
}
```

The response will always have a `200` status if successful or a `404` if the mediator specified by the urn cannot be found. The response body will contain the latest mediator config that has been set on the OpenHIM-core server only if the config has changed since the last time a heartbeat was received from this mediator. Otherise, the response body is left empty.

This endpoint can only be called by an admin user.

Set mediator config

Sets the current configuration values for this mediator. The received configuration must conform to the configuration definition that the mediator defined in its registration message.

`POST /mediators/:urn/config`

where `:urn` is the `urn` property of the mediator that is sending in the heartbeat.

with an http body of:

```
{
  paramName: "value",
  paramName: "value"
}
```

The response will have an http status code of `201` if successful, `404` if the mediator referenced by urn cannot be found and `400` if the config supplied cannot be validated against the configuration definition supplied in the mediator registration message.

This endpoint can only be called by an admin user.

Install mediator channels

Installs channels that are listed in the mediator's config (`defaultChannelConfig` property). This endpoint can install all listed channels or a subset of channels depending on the post body the of request.

`POST /mediaotrs/:urn/channels`

where `:urn` is the `urn` property of the mediator that is installing the channels.

with an http body that contains a JSON array of channel names to install. These names must match the names of channels in the mediators `defaultChannelConfig` property. If no body is sent, all channel are added by default.

e.g.

```
[ 'Channel 1', 'Channel 2' ]
```

The response will be an http status code of `201` if the channels were successfully created and `400` if you provide a channel name that doesn't exist.

Metrics resource

Global Metrics

`https://<server>:<api_port>/metrics`

`/metrics`

This fetches global load and transaction duration metrics for all channels that the logged in user has permissions to view.

These are fetched from either aggregating transaction data from MongoDB or from an instance of the statd metrics service. This depends on the configuration of the HIM in question.

`/metrics/status`

This breaks down the global load metrics by channel and buy status, where the status, could either be 'Processing', 'Failed', 'Completed', 'Successful', 'Completed with error(s)'

Channel Metrics

‘/metrics/[type]/[channelId]

When [type] is status you may expect a response like this

```
[
   {
      "_id":{
         "channelID":"542530aef4e8c76f482bced9"
      },
      "failed":409,
      "successful":280,
      "processing":0,
      "completed":0,
      "completedWErrors":0
   }
]
```

When [type] is hour, day, or month you may expect a response like this

```
[
   {
      "load":1118,
      "avgResp":48223.69219653179,
      "timestamp":"2015-02-19T00:00:00.000Z"
   },
   {
      "load":1725,
      "avgResp":31724.939194741168,
      "timestamp":"2015-02-20T00:00:00.000Z"
   },
   {
      "load":1710,
      "avgResp":34803.78491859469,
      "timestamp":"2015-02-21T00:00:00.000Z"
   },
   {
      "load":1580,
      "avgResp":19637.350119904077,
      "timestamp":"2015-02-22T00:00:00.000Z"
   },
   {
      "load":1637,
      "avgResp":24750.785830618894,
      "timestamp":"2015-02-23T00:00:00.000Z"
   },
   {
      "load":1704,
      "avgResp":31745.67877786953,
      "timestamp":"2015-02-24T00:00:00.000Z"
   },
   {
      "load":1828,
      "avgResp":41017.4289276808,
      "timestamp":"2015-02-25T00:00:00.000Z"
   },
   {
      "load":689,
      "avgResp":37255.648590021694,
      "timestamp":"2015-02-26T00:00:00.000Z"
   }
]
```

If you have any questions that are not covered in this guide, please submit an issue with the 'documentation' label and we will strive to add it to this page.

Keystore resource

The keystore resource allows you to set and fetch the server certificate and key and set and query trusted certificates.

Get the current HIM server cert

`GET keystore/cert`

returns 200 ok with `{ subject: '', issuer: '', validity: '', cert: '<pem string>' }`

Gets the array of trusted ca certs

`GET keystore/ca`

returns 200 ok with `[ { subject: '', issuer: '', validity: '', cert: '<pem string>' }, ... ]`

gets a ca cert by its _id

`GET keystore/ca/_id`

returns 200 ok with `{ subject: '', issuer: '', validity: '', cert: '<pem string>' }`

Sets the HIM server key

`POST keystore/key`

data `{ key: '<pem string>' }`

returns 201 ok

Sets the HIM server cert

`POST keystore/cert`

data `{ cert: '<pem string>' }`

returns 201 ok

Adds a cert to the list of ca certs

`POST keystore/ca/cert`

data `{ cert: '<pem string>' }`

returns 201 ok

Removes a ca cert by its _id

`DELETE keystore/ca/_id`

returns 200 ok

Queries the validity of the server cert and private key

`GET keystore/validity`

return 200 ok with `{ valid: true|false}`

Logs resource

The logs resource allows you to get access to the server logs. This resource is only accessible by admin users and only works if you have database logging enabled (This is enabled by default).

Get logs

`GET logs?[filters]`

Fetches server logs. You may apply a number of filters to fetch the logs that you require. By default the logs with level `info` and above for the last 5 mins with be returned. The logs will be returned as an ordered array with the latest message at the end of the array.

A maximum of 100 000 log messages will ever be returned. So don't make unreasonable queries as you won't get all the results (hint: use pagination).

The following filters are available:

- `from` - an ISO8601 formatted date to query from. Defaults to 5 mins ago.
- `until` - an ISO8601 formatted date to query until. Defaults to now.
- `start` - a number n: the log message to start from, if specified the first `n` message are NOT returned. Useful along with limit for pagination. Defaults to 0.
- `limit` - a number n: the max number of log messages to return. Useful along with `start` for pagination. Defaults to 100 000.
- `level` - The log level to return. Possible values are `debug`, `info`, `warn` and `error`. All messages with a level equal to or of higher severity to the specified value will be returned. Defaults to `info`.

The logs will be returned in the following format with a `200` status code:

```
[
  {
    "label": "worker1",
    "meta": {},
    "level": "info",
    "timestamp": "2015-10-29T09:40:31.536Z",
    "message": "Some message"
  },
  {
    "label": "worker1",
    "meta": {},
    "level": "info",
    "timestamp": "2015-10-29T09:40:39.128Z",
    "message": "Another message"
  }
  // ...
]
```

For example a sample request could look like this:

```
https://localhost:8080/logs?from=2015-10-28T12:31:46&until=2015-10-28T12:38:55&limit=100&start=10&lev
```

Server uptime

`GET heartbeat`

returns 200 ok with `{ master: <core-uptime>, mediators: { <urn>: <mediator-uptime> ... }}`

Returns the server uptime in seconds. Includes a list of all registered mediators and if heartbeats have been received for them, will include their uptimes as well.

Note that this is a public endpoint that does not require any authorization. It is convenient for integrating with external monitoring tools.

5.6 Contributing

If you would like to contribute to either of the OpenHIM projects (openhim-core or openhim-conbsole) then feel free to fork either repository and send us a pull request. The maintainers will strive to review the code and merge it in if all is well.

If you don't want to submit code but you notice an issue with either project, please file it under github issues:

- Submit a core issue
- Submit a console issue

Thanks for contributing! :)

CHAPTER 6

Implementations

6.1 OpenHIE

OpenHIE is an initiative that aims to provide a reference architecture and workflow specifications for sharing health information between point of care systems in low resource settings. It aims to be standards-based and open such that components of the architecture can be swapped out as is necessary. OpenHIE is made up of a number of sub-communities that each aim to discuss a particular component of the architecture. Each community maintains a reference implementation of their particular component.

For more information see the OpenHIE website.

6.1.1 How the OpenHIM is used

The OpenHIM acts as a reference implementation of the interoperability layer component within the OpenHIE architecture. To learn more about this component please see the interoperability layer community wiki.

6.2 MomConnect

MomConnect is a National Department of Health (NDoH) initiative to use cellphone SMS technology to register every pregnant woman in South Africa.

Once enrolled the system will send each mother stage-based messages to support her and her baby during the course of her pregnancy, childbirth and up to the child's first birthday.

The system will also be used to provide feedback (rating, compliments and complaints) about public health services to a central communication centre.

MomConnect aims to strengthen demand and accountability of Maternal and Child Health services in order to improve access, coverage and quality of care for mothers and their children in the community.

The above description is adapted from the RMCHSA website.

6.2.1 How the OpenHIM is used

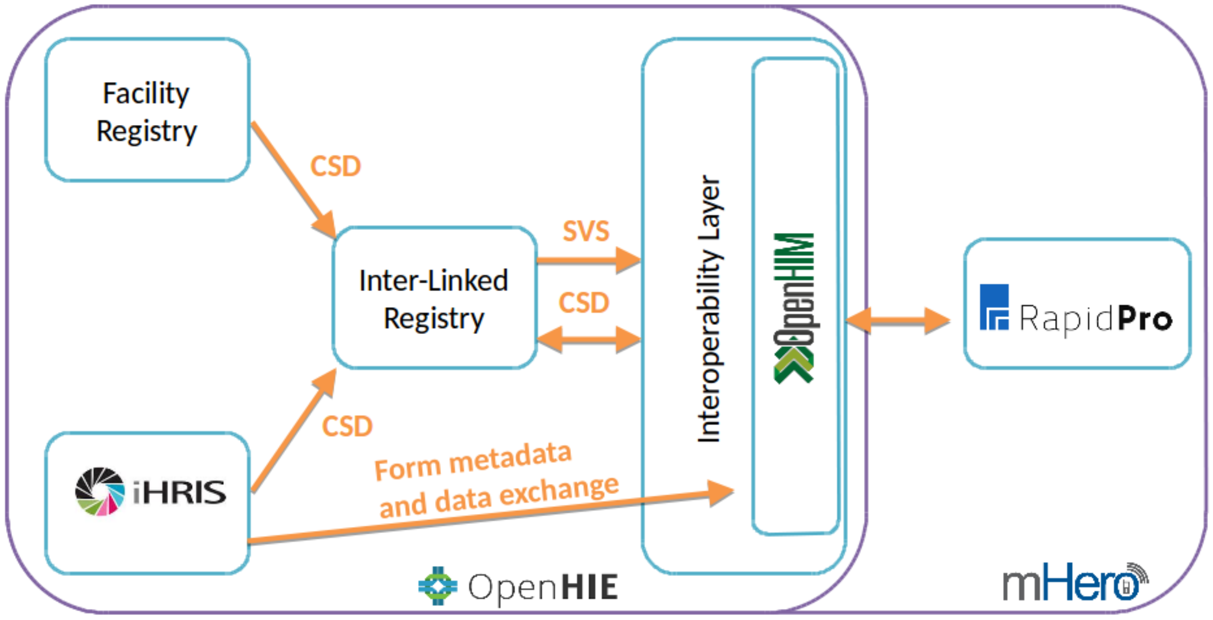
The OpenHIM is used to provide security and visibility into the MomConnect HIE. It also provides a number of orchestration and transformation service to enable pregnancies to be registered correctly in a pregnancy register. Alerting and reporting services are also provided to ensure that the HIE is running smoothly on a day-to-day basis.

6.3 mHero

Mobile Health Worker Electronic Response and Outreach (mHero) is one way to harness the well-known power of mobile technology to reach frontline health workers. It combines data from multiple sources, such as a facility registry and a health worker registry, to enable targeted messaging directly to Health Workers. The messaging workflows that it enables provides an unprecedented link directly to those health workers that are in need of support.

For more information please see the mHero website.

6.3.1 How the OpenHIM is used



Within the context of mHero, the OpenHIM performs a few vital functions.

- It triggers the synchronization between RapidPro and the OpenInfoMan.
- It provides visibility into the messages being exchanged. This allows the user to ensure that the data exchange is occurring correctly.
- It ensures that the communication between components occurs securely and it logs the transactions for historical and audit purposes.
- It provides authentication and authorisation mechanisms to control access to the OpenInfoMan documents

The OpenHIM provides polling channels to trigger the synchronization between RapidPro and the OpenInfoMan. These polling channels execute periodically and trigger an mHero mediator which in turn pulls data out of the OpenInfoMan and pushes it into RapidPro. To learn more about polling channels please see the OpenHIM docs here.

The OpenHIM provides a web console that enables the user to view these synchronization message. This enables any problems to be debugged effectively and provides confidence that the synchronization is working effectively.

The OpenHIM was designed to protect an HIE by providing mechanisms to secure transactions between various components of the HIE. It can ensure that requests that access certain OpenInfoMan documents come from known and authorised sources.

Within mHero, the OpenInfoMan contains a number of documents which contain health worker and facility information. The OpenHIM prevents unauthorised access to these documents by implementing a role-based access control mechanism. This allows documents with sensitive information to be secured and documents with non-sensitive information to be as open and accessible as necessary.

CHAPTER 7

How to's

7.1 How to do an OpenHIM-core release

This page describes the steps to follow to do an OpenHIM release. Make sure you are on the `master` branch and it is fully up-to-date before beginning this process.

1. `npm shrinkwrap`
2. `git add npm-shrinkwrap.json`
3. `git commit -m"Added shrinkwrap for x.x.x release"` - replace x.x.x with the actual release version.
4. `npm version (major|minor|patch)` - choose one according to semver.
5. `npm publish`
6. `git rm npm-shrinkwrap.json`
7. `git commit -m"Removed shrinkwrap for continued development"`
8. `git push origin master`
9. `git push origin vx.x.x` - push the tag that 4 created.
10. Create a new github release using the tag created in 4 above, that includes the release notes.
11. Build a debian package and upload it to launchpad. Follow the instructions here.
12. Write a blog post on openhim.org that includes the release notes.

7.1.1 Support Releases

From time to time a support release may be required for critical security issues or bugs. When this happens, a support branch should be created in order to support that particular version.

If the branch doesn't exist, create it from the latest tag for a particular release:

- `git checkout -b support-vx.y vx.y.z` else if the branch exists, simply check it out and continue from there
- `git checkout support-vx.y`

Ideally fixes should first be developed separately and merged into master. They can then be cherrypicked for the support release:

- `git cherry-pick bd68fe1c8cf81cbef2169414ce8440a7a2c69717`

Although this may not always be possible, in which case the fixes can be added manually.

The shrinkwrap can be left in place on a support branch. If however the support fixes include package updates, rerun the `npm shrinkwrap` command in order to update.

When all fixes have been applied, test thoroughly and create a new release as per normal:

1. `npm version (major|minor|patch)` - increment the patch version.
2. `npm publish`
3. `git push origin support-vx.y`
4. `git push origin vx.y.z` - push the new tag
5. Create a new github release

When a particular version is no longer supported, its support branch should be deleted.

7.2 How to do an OpenHIM-console release

This page describes the steps to follow to do an OpenHIM console release. Make sure you are on the master branch and it is fully up-to-date before beginning this process.

1. `npm version (major|minor|patch)` - choose one according to semver.
2. `git push origin master`
3. `git push origin vx.x.x` - push the tag that 2 created.
4. Run `grunt` then `tar cvzf openhim-console-vx.x.x.tar.gz -C dist/ .`
5. Create a new github release using the tag created in 3 above that includes the release notes and attach the tar.gz distribution created in 4.
6. Write a blog post on openhim.org that includes the release notes.

7.3 How to pre-package an offline OpenHIM-core release

Sometimes it's necessary to install the HIM in a locked down environment, e.g. on a corporate controlled server with a firewall that blocks npm, or in an environment with poor internet access. In these cases it would be useful to prepackage the HIM suitable for installation in such environments. The following instructions detail how to do this using offline-npm.

- Install offline-npm: `npm install -g offline-npm`
- Checkout the relevant release of the HIM:
  - `git clone https://github.com/jembi/openhim-core-js.git`
  - `cd openhim-core-js`
  - `git checkout vx.y.z`
- Build the HIM: `npm install`
- Enabled offline-npm: `offline-npm -a`

Next, edit `package.json` and change the line

```
"prepublish":  "./offline/offline-npm --prepublish ; grunt build",
```

removing the `grunt build` command:

`"prepublish": "./offline/offline-npm --prepublish ;",`

(there is an issue with grunt build not working after adding offline-npm)

- Finally create the package: `npm pack`

This should should create a package `openhim-core-x.y.z.tgz`. You can now copy this package onto the server and install it using the command: `npm install -g openhim-core-x.y.z.tgz`.

7.4 How to run the OpenHIM using vagrant

If you're a developer, or would just like to get the OpenHIM up and running for testing purposes, the quickest way to do so is to fire up a Vagrant instance.

7.4.1 Steps

- Setup Vagrant on your system. Note that you'll also have to install VirtualBox.
- Clone the repo
- (if necessary) `sudo apt-get install git`
- `git clone https://github.com/jembi/openhim-core-js.git`
- Launch the instance
- `cd openhim-core-js/infrastructure/deployment/env`
- `vagrant up`

And that's it! Your Vagrant instance should now be up and running. You can access it by running the command `vagrant ssh`. The OpenHIM itself will be available in the `/openhim-core-js` directory. You can proceed as follows in order to run it:

```
vagrant ssh
> cd /openhim-core-js
> grunt build
> node --harmony lib/server.js
```

If you would like to run the console as well, the easiest way is to fire up another vagrant instance [in another terminal]:

```
git clone https://github.com/jembi/openhim-console.git
cd openhim-console/infrastructure/deployment/env
vagrant up
vagrant ssh
> cd /openhim-console
> grunt serve
```

Note that the vagrant instances have port forwarding enabled, so to access the console you can do so by just navigating to **http://localhost:9000** in your browser on the system that's running the vagrant instance, not the instance itself (which you would struggle to do anyway!).

When you're done you can dispose of an instance by running `vagrant destroy` (not ssh'd into the vagrant instance).

7.5 How to run the OpenHIM on startup

To help you get the OpenHIM server running on boot we supply a upstart config file (good for Ubuntu or other system that use upstart). Install the upstart config by doing the following:

```
wget https://raw.githubusercontent.com/jembi/openhim-core-js/master/resources/openhim-core.conf
sudo cp openhim-core.conf /etc/init/
```

Then run start the server with:

```
sudo start openhim-core
```

It will automatically startup on reboot.

If you require custom config you will have to edit `openhim-core.conf` to add in the `--conf` parameter pointing to your external config file.

7.6 How to export/import Server Configuration

Note: This can now be done directly from the OpenHIM console which may be easier.

7.6.1 Exporting

Follow the below steps to export and import the server metadata configuration. By default, the Users, Channels, Clients, ContactGroups and Mediators collections will be exported. Copy the file openhim-configuration-export.sh to a folder where you wish your export to be saved. Run the shell scrip by executing the following command: `./openhim-configuration-export.sh`

Your exported collections should be located in the folder structure '/dump/openhim/'.

7.6.2 Importing

To import you data successfully ensure that you are in the correct folder where the dump files are located. Execute the below command to import your collections. `mongorestore --db openhim dump/openhim`

NB! if you have changed your database name, then do so for the export/import as well. NB! Please ensure that you stop the server before exporting and importing.

7.7 How to setup a basic cluster

The OpenHIM Core is designed to be horizontally scalable. This means that if you need better performance, you can easily fire up more core instances. In this tutorial we will look at setting up a small, basic cluster of core instances.

7.7.1 Background

The OpenHIM Core is built in Node.js and it is important to note that node uses a single threaded model. The threading model is designed for I/O bound processes - perfect for the OpenHIM - however this can lead to a single instance of core quickly becoming a bottleneck on very high transaction loads. In addition, a core single instance wouldn't take advantage multiple cores on a CPU. On a dedicated server it thus recommended that one OpenHIM instance is used for every available CPU core. In addition it would of course be useful to be able to run multiple core instances on multiple servers as well.

7.7.2 Clustering on a single server

Luckily, since v1.2.0 of the OpenHIM, clustering on a single server is supported out of the box. So, setup is really easy. All you need to do is run the OpenHIM with a flag to let it know you want to cluster:

```
$ openhim-core --cluster=N
```

N is the number of instances that you want to run (eg, 2, 4 or 6) or it can be the special value of 'auto' where the OpenHIM will determine how many core your server has and run that many instances. Each instance that the OpenHIM starts shares the same ports and the OpenHIM will share load between all of these instances.

7.7.3 Clustering over multiple servers

The approach we'll be taking towards scaling out core to multiple server is also really straight forward! We'll simply fire up 4 instances on separate servers and setup load-balancing between them. For this tutorial we'll look at using LVS on Ubuntu for load-balancing, but other options exist as well:

- NGINX - A very powerful load-balancer and web server. Note however that TCP load-balancing is only supported in their NGINX Plus commercial product. However http load-balancing can still be used, but can become more complex if you want to use https channels on the HIM.
- node-harmony
- loadbalancer

First, install the OpenHIM on each server and grab a copy of the config file is you wish to use a non-default configuration:

```
$ npm install -g openhim-core $ wget https://raw.githubusercontent.com/jembi/openhim-core-
```

Next, startup each instance on each server, you may also use the –cluster option if you choose to.

```
$ nohup openhim-core --cluster=auto
```

Now we can setup the load-balancer. If not already available, install the LVS Admin tool on the server that will act as you load balancer:

```
$ sudo apt-get install ipvsadm
```

Now we'll setup round-robin balancing and add each node to the cluster:

```
$ sudo ipvsadm -A -t 10.0.0.1:5000 -s rr
$ sudo ipvsadm -a -t 10.0.0.1:5000 -r 10.0.0.2:5000 -m
$ sudo ipvsadm -a -t 10.0.0.1:5000 -r 10.0.0.3:5000 -m
$ sudo ipvsadm -a -t 10.0.0.1:5000 -r 10.0.0.4:5000 -m
$ sudo ipvsadm -a -t 10.0.0.1:5000 -r 10.0.0.5:5000 -m
```

Replace the 10.0.0.x addresses with your servers' IP addresses. You can also follow these steps for the HTTP ports and the API ports (although another good strategy with regard to the API is to run another dedicated core instance and point the Console there). And that's it! Try running several transactions against your server on HTTPS - they should be load-balanced between the four instances AND each instance will itself be clustered on the server that it is running! This should allow you to handle MASSIVE load with ease.

7.8 How to manually install on Ubuntu 14.04 Trusty

The following is a quickstart tutorial to help guide you through the steps required for a new OpenHIM installation on a clean Ubuntu 14.04 instance.

This quickest an easiest way to install the OpenHIM on ubuntu is to use our deb package. This will install both he OpenHIM core and console on your server. See details on how to do this here.

If you would like to install the OpenHIM manually, read on.

7.8.1 Install Node.js 0.12.x

As per https://nodesource.com/blog/nodejs-v012-iojs-and-the-nodesource-linux-repositories The first required dependency is Node.js. You should at least be running version 0.12, which isn't available in the official Ubuntu repositories in 14.04. Therefore we need to use an alternative source:

```
$ curl -sL https://deb.nodesource.com/setup_0.12 | sudo bash - sudo apt-get
install nodejs
```

7.8.2 Install MongoDB 3.0

As per http://docs.mongodb.org/master/tutorial/install-mongodb-on-ubuntu Next we need to setup MongoDB. At a minimum version 2.6 is required, but let's get version 3.0:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
$ echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | suc
$ sudo apt-get update
$ sudo apt-get install mongodb-org
```

7.8.3 (optional) SSH User Fix - needed to use mongo client

You may run into an issue when trying to run the mongo command from the commandline via an SSH session:

```
$ jembi@openhim:~$ mongo
Failed global initialization: BadValue Invalid or no user locale set. Please ensure LANG and/or LC_*
```

This can be fixed as follows: Use your favourite text editor to open up `/etc/default/locale` and add the following line:

```
LC_ALL="en_US.UTF-8"
```

or use whichever locale is appropriate. Now log out and back in from your SSH session.

7.8.4 Other prerequisites

Just some final dependencies before we move onto the HIM installation itself:

```
$ sudo apt-get install git build-essential
```

7.8.5 OpenHIM Core

Now that all our dependencies are in place, let's proceed with installing the OpenHIM Core component:

```
$ sudo npm install -g openhim-core
```

This will download and install the latest version of core. Next we'll setup the configuration and an Ubuntu service.

Configuration

Download a copy of the default core config and place it in */etc*:

```
$ wget https://raw.githubusercontent.com/jembi/openhim-core-js/master/config/default.json
$ sudo mkdir /etc/openhim
$ sudo mv default.json /etc/openhim/core.json
```

You can now edit `/etc/openhim/core.json` and configure it as required for your instance.

Setup the HIM core as a service

Download a copy of our default service configuration:

```
$ wget https://raw.githubusercontent.com/jembi/openhim-core-js/master/resources/openhim-core.conf
$ sudo mv openhim-core.conf /etc/init/
```

Next edit `/etc/init/openhim-core.conf` and edit the startup line to look as follows:

```
NODE_ENV=production openhim-core --conf=/etc/openhim/core.json --cluster=auto >> /var/log/openhim-cor
```

Here we're just setting up the startup command to use the configuration in /etc/openhim, as well as enabling automatic clustering; which will take advantage of your available CPU cores.

Run and verify

Now we're ready to startup the HIM Core:

```
$ sudo service openhim-core start
```

You can verify and monitor the instance by looking at the logs:

```
$ tail -f /var/log/openhim-core.log
```

7.8.6 OpenHIM Console

Next we need to setup the OpenHIM Console. Download the latest release from https://github.com/jembi/openhim-console/releases/latest, e.g.:

```
$ wget https://github.com/jembi/openhim-console/releases/download/v1.2.0/openhim-console-v1
```

In this example we downloaded version 1.2.0, but it's a good idea to get the latest that is available. Next we need a web server to host the console; for this tutorial we'll use Nginx:

```
$ sudo apt-get install nginx
```

Now deploy the console:

```
$ cd /usr/share/nginx/html/
$ sudo tar -zxf ~/openhim-console-v1.2.0.tar.gz
```

Next we need to edit `/usr/share/nginx/html/config/default.json` and configure for the HIM core server. Simply set the host and port values to point to the address that the HIM core API will be available from. Note this host needs to be publicly accessible, e.g. the server's domain name or public IP address. When a client uses the HIM console, requests to the core API will be made "client-side" and not from the server. Now we can startup Nginx and start using the HIM:

```
$ sudo service nginx start
```

7.8.7 Fin

The OpenHIM Core and Console should now be up and running! Access the console on http://yourserver and login with **root@openhim.org** using the password: **openhim-password** If there's a login issue, try accepting the self-signed cert in your browser on: *https://yourserver:8080/authenticate/root@openhim.org*

7.9 How to manually install on Windows

The following is a quickstart tutorial to help guide you through the steps required for a new OpenHIM installation on a Windows instance.

7.9.1 Install Node.js LTS

Install the latest LTS version of Node.js from their official site. Note that the OpenHIM only officially supports the LTS edition of node, which is version 4.x at time of writing.

The official process should be suitable for the OpenHIM; simply download and run the installer msi.

7.9.2 Install MongoDB

Install the latest version of MongoDB from their official site

As with Node.js, the official process should be suitable for the OpenHIM. Note however that MongoDB requires some additional steps after running the installer - in particular it would be a good idea to setup MongoDB as a service.

The following guide should help you get fully setup: https://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

7.9.3 OpenHIM Core

Install

To install the OpenHIM Core, launch a Node.js command prompt via **Start > All Programs > Node.js > Node.js command prompt**. From here you can install Core using the following command

```
npm install -g openhim-core
```

You may see some warnings during the install process, especially if you do not have a C++ compiler installed, but this is not a problem and you can ignore these.

Configuration

Create a folder for storing the OpenHIM config, e.g. `C:\OpenHIM` and grab a copy of the default config from github and save it to locally, e.g. `C:\OpenHIM\core.json`. Change this config to suit your needs.

You should now be able to run the OpenHIM Core. In a Node.js command prompt, run the following:

```
openhim-core --conf=C:\OpenHIM\core.json
```

or with whichever file location you chose to create for the config.

Run as a Windows Service

To ensure the OpenHIM runs all the time, we will install it as a Windows Service using NSSM (the Non-Sucking Service Manager)

1. Download NSSM (the Non-Sucking Service Manager)
2. Open the archive and extract the `win32` or `win64` directory (depending on your Windows architecture) to a location on disk, for example `c:\nssm`
3. Add the location `c:\nssm` to your path, so that `nssm` is accessible without knowing and typing the whole path to the file on the command line
4. Open a command window with administrator privileges
5. Type `nssm install openhim-core "C:\Program Files\nodejs\node.exe" "<insert-full-path>\node_modules\openhim-core\bin\openhim-core.js --conf=C:\OpenHIM\core.json"`
6. To capture the log output, type `nssm set openhim-core AppStdout "c:\OpenHIM\stdout.log`
7. To capture the error output, type `nssm set openhim-core AppStderr "c:\OpenHIM\stderr.log`
8. Type `net start openhim-core` to start the service or start it from the service manager.

You're done. You've installed the OpenHIM as a windows service.

7.9.4 OpenHIM Console

A web server will be required to host the OpenHIM Console and in this guide we will use IIS and as an alternative we will also explain how to use Nginx. However any good web server will be suitable, e.g. Apache.

Install IIS

Go to http://www.iis.net/learn/install for articles on how to install IIS for your particular flavour of Windows OS.

If you want to check if IIS is installed, browse to http://localhost in your browser. If an image pops up, then IIS has been installed correctly.

Setup Console

Download the latest Console release and extract the contents into a folder such as `C:\OpenHIM\Console`. Note that you will need to use a utility such as 7-Zip to extract the .tar.gz archive.

Console contains a config file located in `Console\config\default.json`. You will need to edit the `host` and `port` fields to point to the *public* address that the OpenHIM Core is running on. If you are only using the OpenHIM locally, then it is fine to leave the setting on localhost, however if you wish to make the Console accessible to other hosts, you will need to change the setting to either the machine's public IP address or domain name.

Configure the Console for IIS

Create a new site in Internet Information Services Manager. You can name it whatever you want. I'll call it Console in these instructions.

1. Start IIS Manager.

2. In the Connections panel, expand Sites
3. Right-click Sites and then click Add Web Site.
4. In the Add Web Site dialog box, fill in the required fields, for example:
    - Site name: `Console`
    - Physical path: `C:\OpenHIM\Console`
    - Port: Make sure the port is something other than 80, as this will conflict with "Default Web Site" in IIS

Alternative Web Server Instructions

Install Nginx

A web server will be required to host the OpenHIM Console and in this guide we will use Nginx. However any good web server will be suitable, e.g. Apache or IIS.

As per this guide, download and extract the Nginx windows binary. You don't need to start nginx yet however.

Setup Console

Download the latest Console release and extract the contents into a folder such as `C:\OpenHIM\Console`. Note that you will need to use a utility such as 7-Zip to extract the .tar.gz archive.

Console contains a config file located in `Console\config\default.json`. You will need to edit the `host` and `port` fields to point to the *public* address that the OpenHIM Core is running on. If you are only using the OpenHIM locally, then it is fine to leave the setting on localhost, however if you wish to make the Console accessible to other hosts, you will need to change the setting to either the machine's public IP address or domain name.

Next locate the Nginx configuration file `<nginx location>\conf\nginx.conf` and change the root context to point to the Console:

```
location / {
    root   C:\OpenHIM\Console;
    index  index.html index.htm;
}
```

Also change any other settings as required, e.g. port numbers.

Now you can startup Nginx from a command prompt by running:

```
cd <nginx location>
start nginx
```

7.9.5 Fin

The OpenHIM Core and Console should now be up and running!

Access the console on http://yourserver: and login with **root@openhim.org** using the password: **openhim-password**

7.10 How to build the documentation

To build the documentation execute the commands below from the 'docs' directory:

1. `$ sudo pip install -r requirements.txt`
2. `$ make html`
3. (optional) `$ sphinx-autobuild . _build/html` - to host locally and automatically build the docs on change.
4. When you push changes to origin they will automatically be built on readthedocs.org.