Design of the Interoperability Layer core using Node.js

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

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.

General Overview

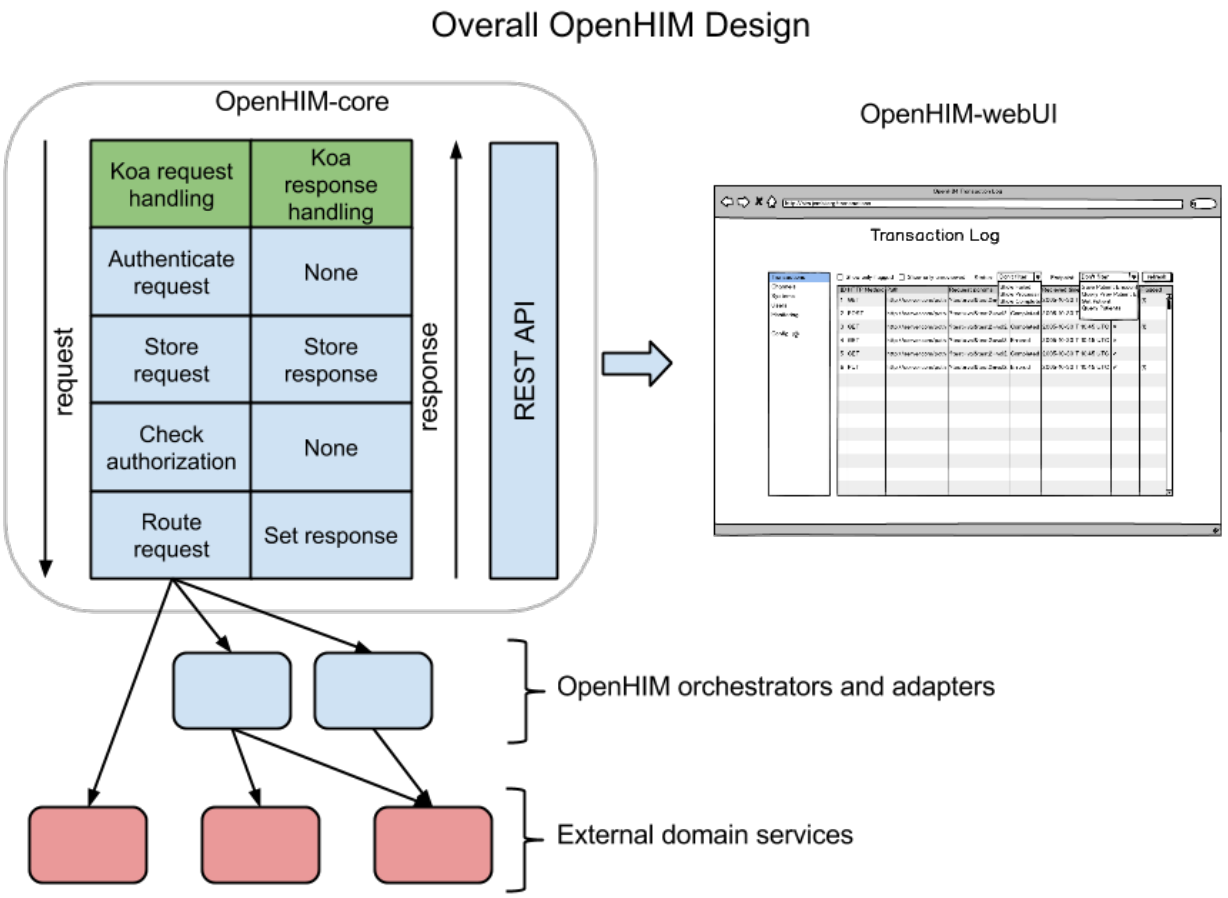
The Koa framework provides an easy way to modify HTTP request and responses as they are being processed. Each step that the OpenHIM needs to perform can be written as Koa middleware. Each middleware can handle a different aspect of processing that the OpenHIM need to perform such as authentication, authorization, 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.

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.



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:

application.json

```
{
        "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.

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.

transaction.json

```
{
        "_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"
        }
}
```

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.

channels.json

```
[
        {
                "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" }
                ]
        }
]
```

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 is define in using RAML. You can view the details of the API here or you can view the raw RAML code 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"]
}
```