2016

DEVELOPMENT & INTEGRATION OF FHIR WITH OPEN-SOURCE EMR & RELATIONAL DATABASES

Timothy Van Wagner
Michigan Technological University, tavanwag@mtu.edu

DEVELOPMENT & INTEGRATION OF FHIR WITH OPEN-SOURCE EMR & RELATIONAL DATABASES

By

Timothy Van Wagner

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Medical Informatics

MICHIGAN TECHNOLOGICAL UNIVERSITY

2016

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Medical Informatics.

School of Technology

Thesis Advisor:	*Guy C. Hembroff*
Committee Member:	*Dr. Donald Joseph Peck*
Committee Member:	*Dr. Gowtham S*
Department Chair:	*Guy C. Hembroff*

Contents

List of Figures

List of Abbreviations

FHIR	Fast Healthcare Interoperability Resources
SQL	Structured Query Language
JSON	JavaScript Object Notation
JDBC	Java DataBase Connectivity
XML	eXtensible Markup Language
ETL	Extract, Transform, Load
HL7	Health Level 7
CDA	Clinical Document Architecture
EMR	Electronic Medical Record
EHR	Electronic Health Record
DSTU2	Draft Standard for Trial Use version 2
ONC	Office of the National Coordinator
REST	Representational State Transfer
API	Application Program Interface
PHP	Hypertext Preprocessor
HAPI	HL7 Application Programming Interface

Abstract

This project looks into the development and implementation of an open source EHR software, using a FHIR interface to the system for data exchange. The EMR system used was OpenEMR, which is an open source EMR system that uses MySQL as the database backend combined with Apache and PHP to provide a user interface. The FHIR interface uses HAPI-FHIR which is an open source framework and libraries available for JAVA. Using JAVA, a RESTful API is created in order to respond to requests, and to also receive FHIR messages using the information stored in the two SQL databases, OpenEMR database and the FHIR database.

Introduction

In health care there is a large gap in interoperability. Proprietary systems in different organizations make it difficult to exchange patient data between the two or more entities. It also makes it challenging to have devices access data in one's database if it does not use the same data structure and definitions. An example would be how different databases store and read date information. If database A stores a date value for when a patient dies, it would most likley have something similar to a deceased_date field to store this in the database. When a patient is still alive, this field will not have a real date stored in it. In some databases this could show up as "0000-00-00" to indicate the patient is not dead and in another database the same type of value could show up as "null" to indicate a patient is still alive. This difference in values is one reason it is difficult to exchange information between organizations and also why a standard format is needed for exchange.

Fast Healthcare Interoperability Resources, FHIR, pronounced fire, is the next step in creating a common standard to healthcare and devices connected and exchanging

data from multiple sources. Health data transfer through FHIR is also considered mobile friendly compared to the previous versions of HL7 messaging systems. "FHIR is encoded in a format that is much more compact than the code currently used to exchange clinical data, and can easily live on mobile devices."[1] The use of FHIR reference values within its resource allows information to be gathered from multiple locations and organizations.

The goal of the project is to show the viability of incorporating a FHIR server into an open source EMR system. The incorporation of a RESTful API into an open source EMR system can help decrease the cost of implementation while increasing interoperability between organizations. Time of implementation should also be decreased from having a FHIR interface that can be setup quickly with an existing application database. The EMR system used will be OpenEMR. This is a PHP based EMR system that uses MySQL for the backend database. This means that the system can be run on many different platforms, including Windows, Linux and Mac computers. This program is also an ONC certified program. Having an open source EMR system means that it will be easier to modify for the purposes of this project. In order to apply a FHIR server to the EMR system, HAPI-FHIR was chosen. HAPI-FHIR is an open source JAVA framework for creating and implementing a server and/or client using the FHIR standard. HAPI-FHIR is using FHIR DSTU2. This is the most recent FHIR structure as of the writing of the paper and will be the version used in this project.

FHIR

FHIR was created to be a framework which can be leveraged to connect health data. It combines the features of HL7 v2, v3 and CDA to create one health information exchange standard. FHIR is a well-thought out standard, building on lessons learned from the past and takes the best features from other standards. The FHIR framework utilizes different *resources* for health record classification. These *resources* define different use cases of data that may need to be exchanged, and can be accessed by querying information from a FHIR server through a pull method or the server can push information to a different server as new data is created. If a record needs to be obtained and displayed, it will access the FHIR server with a query for a particular resource. This resource can be returned in two different formats, depending on the application that made the request. It can be returned in an XML format. XML will be supported by all FHIR server implementations. The server can also return the resource in a JSON format. The FHIR server does not need to support JSON and is left up to the implementer of the server.

With the ability to query different types of resources, the information returned to a user or provider does not require a large document as with previous standards. FHIR can be lightweight which makes it mobile friendly. That means each resource accessed through a FHIR API can be just a small portion of a patient's story. Many

different resources can be retrieved in order to tell the whole story or just a small set of resources can be retrieved to access the data quickly. Multiple resources are gathered through a field in the resource called references. These are the URLs of information linked to the resource that is held as another resource. These references can be internal to the server that presented the first resource or external on a different organization's server. These references can either be displayed as the URL only to the user or can be displayed as all the information returned by the resource. This all depends on the client application and what it wants to do with the reference information.

FHIR resources

FHIR does not use one large document to transmit information. It is broken down into resources. A resource is a small unit of data which is representative of the clinical data it holds. The resource can be addressed by a URL that is specific to it. Each resource definition has a defined set of structured data. A resource could be compared to a table in a SQL database. The table would contain information on one small piece of the overall picture of a patient's health status and have a key that will connect it to the other tables such as the Medical Record Number (MRN).

The structure of a resource follows a set of levels. On the top level is the metadata

for the resource, which talks about the message itself like the transmission time or when the message was last updated. Below, is the narrative data. Which is human readable giving a description presented in the resource. The human readable section takes on the form of some HTML code that can be displayed in a Web browser. The next level is the extension data. This is any data added to fit the requirements of the organization that is not met in the base structure of the resource. The bottom level is the defined structure data. This is the data that has been defined to be a requirement for the resource.

There are a handful of defined interactions or operations that can be performed on each resource when connecting to a RESTful API server. Resources have their own ID that is local to the organization's FHIR server. The URL points to the ID of the resource on that server. A resource can have many different identifiers, each one with a unique value to itself and unique under different organizations. This means that an external identifier on the FHIR server needs to also contain a system or domain in which the identifier is considered unique. A patient can be found using a unique value that is known for the system or domain if the RESTful API allows for searching of resources by *identifier*. Otherwise the local ID for the FHIR server needs to be known or another *search-able* value on the server.

Method

Tools Used

The FHIR database was built using Power Architect. Power Architect allows for a visual representation of tables and columns to be translated into the syntax needed for each type of SQL system [2]. This allows the database schema to be applied to other relational databases such as PostgreSQL, Oracle, MSSQL and MySQL. There are slight differences in syntax between the different database management systems and using a tool to cover all of them reduces the time required to work on multiple databases. Most of the differences deal with data types names. For example, MySQL uses `LONGBLOB` data types to store a large amount of data in a field while in MSSQL the equivalent would be an `IMAGE` data type. Figure 1 shows the tables and relationships created in Power Architect to interact with the FHIR server for the patient resource. There are two tables, `patient_Archive` and `patientDBmap`, that do not show any relationships to the other tables. These tables are not actually used for

the information stored in a FHIR resource but are for the management of the patient resource on the FHIR server.

Database Tables

Each FHIR resource has multiple tables. These tables are associated with one *main* table for the resource. The patient resource main table is shown in Figure 2. Any columns that are contained in the main table only occur once for a resource, according to the FHIR standard [3]. Tables that branch off of the main resource table have the possibility of containing zero or infinite values for each column depending on the particular value. FHIR defines how many of each value can be possible in every resource. For example, in a patient resource there can be an infinite number of contacts but the contact can only have up to or less than one name value. Figure 3 displays what the FHIR standard allows for contact information in the patient resource in the third column using "0..1" or "0..*", * indicating an infinite number of values allowed. This is the reason for having multiple tables branch off the main, to allow for this infinite number of possible values. The main resource table contains a column, `localID`, which is used to identify the specific resource to the server. The `localID` is a unique value to the main table only. A local identifier is required by the FHIR standard for all resources and has to be a unique value to the FHIR server that is providing information to clients and/or applications. The FHIR standard does

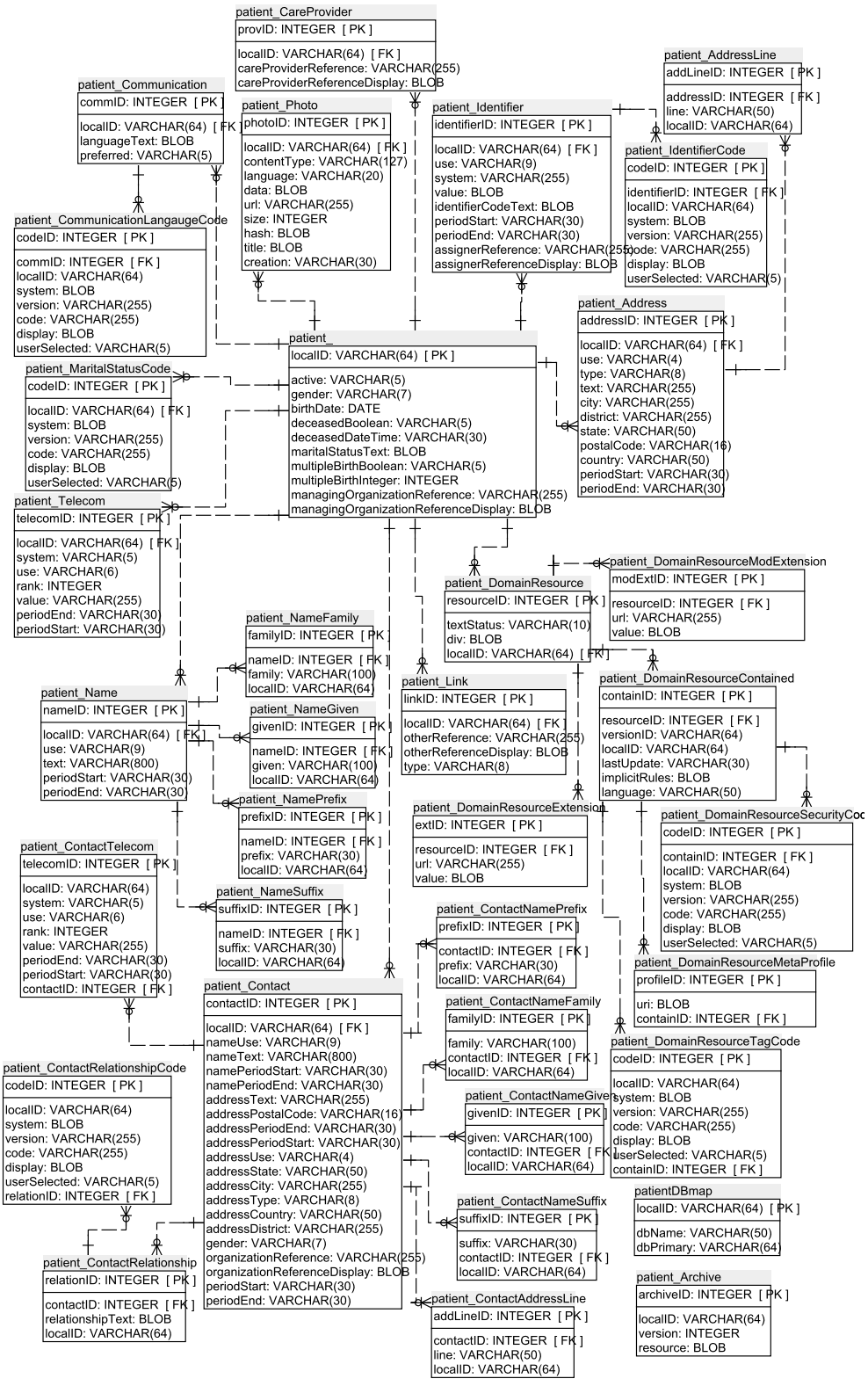

Figure 1: Patient resource tables and relationships for FHIR server

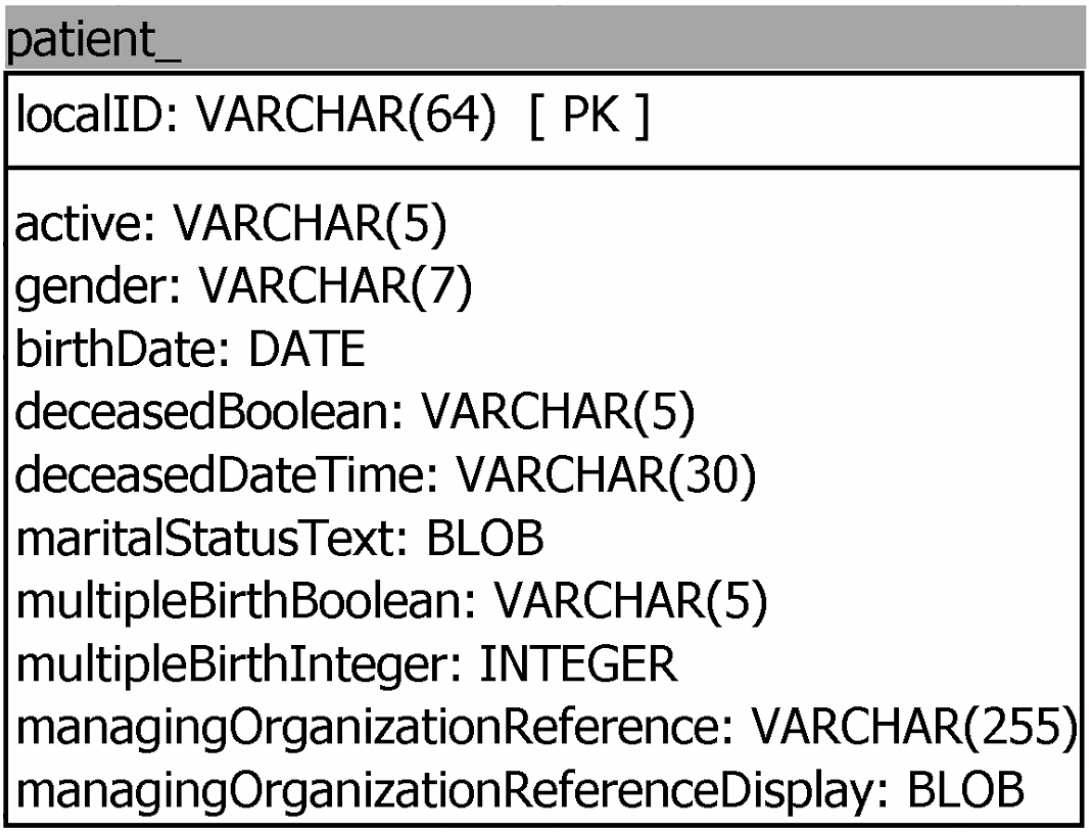

Figure 2: SQL table of main patient resource

not require that the column be named `localID`, that is left up to the implementer. The `localID` on the server is created by calling the `createLocalID` method from the JAVA `Identifier` class in the server. The `createLocaID` method looks up the largest `localID` value in the database and then increments up. If the `localID` is 1 then the `localID` to create the new record is 2. If the `localID` is A then the `localID` to create the new record is B. This is just a simple example but since the FHIR standard requires the possibility of using 64 alphanumeric characters, this allows for 36^{64} possible unique values (about 4 duotrigintillion). Since the other tables for the resource branch off from the main, `localID` is used in every table to associate itself

Figure 3: FHIR patient resource contact structure. Adapted from https://www.hl7.org/fhir/patient.html Copyright HL7.org 2011+

to the unique resource. This isn't to say that the `localID` is a foreign key in every table, `localID` is a foreign key only to tables that are directly connected to the main resource table. Tables that are indirectly connected contain foreign keys that link to tables that are between itself and the main table. These indirectly connected tables still contain the `localID` of the resource it is associated with in order to help those that may be dealing with the database backend of the RESTful server.

There may be many tables with the same column and data types for a individual FHIR resource such as code tables. The tables will be named differently but the structure is the same as others. These tables hold what is considered a codeable concept in the FHIR standard. In the backend database there are many tables that have the same

columns and data types but have different table names. The reason for this is that when someone is working with the FHIR database they will be able to easily know what the codes they are looking at are associated with inside of that particular FHIR resource. The tables will always have the resource name at the beginning followed by an underscore and then information of what the data it holds for the resource. Not only does this allow you to know what you're working with but it will also reduce the size of the table. Queries will not have to be run against codeable concept tables that contain information for all FHIR resources and data types.

The FHIR standard links resources together by using references. References contain two parts, the path to the resource and a display which describes the reference in human readable form. Neither one is a required data field. For every FHIR resource a reference is contained in two separate columns. Any reference that ends with the word "reference" in the SQL table is going to contain the URL of a FHIR resource that the other FHIR resource is referencing. The second column ends with the word "display". This is contained in a `BLOB` data type to allow, in MySQL, enough storage space in the containing field to sufficiently describe the reference. Boolean values are stored as a `VARCHAR(5)` data type. This was to simplify adding the value to the RESTful server. Using the HAPI-FHIR framework, boolean values are created by typing either true or false in the JAVA code. Storing the value as a Boolean data type in the database means that when an SQL query is made the return value to the server ends up being a 1 or 0. The true or false `VARCHAR` values were used to simply

reduce the amount of logic used in the server's JAVA code because the stored values as a string can be used directly with the HAPI-FHIR framework. Hash values and data columns in the FHIR resource are stored as `BLOB` values in order to keep the database from applying any type of formatting to the information and also to allow for sufficient storage space of the values. Any FHIR `dateTime` value is stored as a `VARCHAR(30)`. This is to allow storage of the date and time along with the time zone in the format that is specified by the FHIR standard and still be able to have this database be used across multiple types of relational databases.

RESTful API using JAVA

The RESTful API was written in JAVA using HAPI-FHIR. HAPI-FHIR is a set of JAVA libraries that can be used to quickly write JAVA code and apply FHIR to applications. HAPI-FHIR provided a quick and simple way to start a RESTful API. This way most of the focus could be spent on parsing the information correctly in order to store it in the database created for this project and to create the logic to work with the application database. The FHIR server currently supports the ability to read, search and update patient information from an attached application database and also the FHIR database.

HTTP GET

Figure 4 shows the logic when the FHIR server receives an HTTP GET command. A HTTP GET command covers different types of interactions on the server. On this particular server it covers read and search. As of now the server does not support the ability to `vRead`, which is a version read of older resources. When a HTTP GET command is received the server determines if it is a search or read interaction by checking if a local ID for the server was provided or if there is no local ID, it checks what parameters are provided for a search. When a local ID is present the FHIR server will attempt to look up that particular ID from the `patientDBmap` table located in the FHIR database. If a match for the requested local ID is not found in this table, an `OperationOutcome` is returned to the requester with a "Resource Patient/ID is not known" message as shown in Figure 5. If a match is found in `patientDBmap` table then the appropriate database name is returned from this search to tell the server where to look. In the event that "FHIR" is returned from `patientDBmap`, the server will lookup up the information for the record from the FHIR database and only the FHIR database. This simply means that the information for the patient has not been pulled into the application that the FHIR server is attached to. The information only resides in the FHIR database. When "OpenEMR" or the name of the attached application being used is returned, the server will pull information from both the attached application database and also the FHIR server database. The information

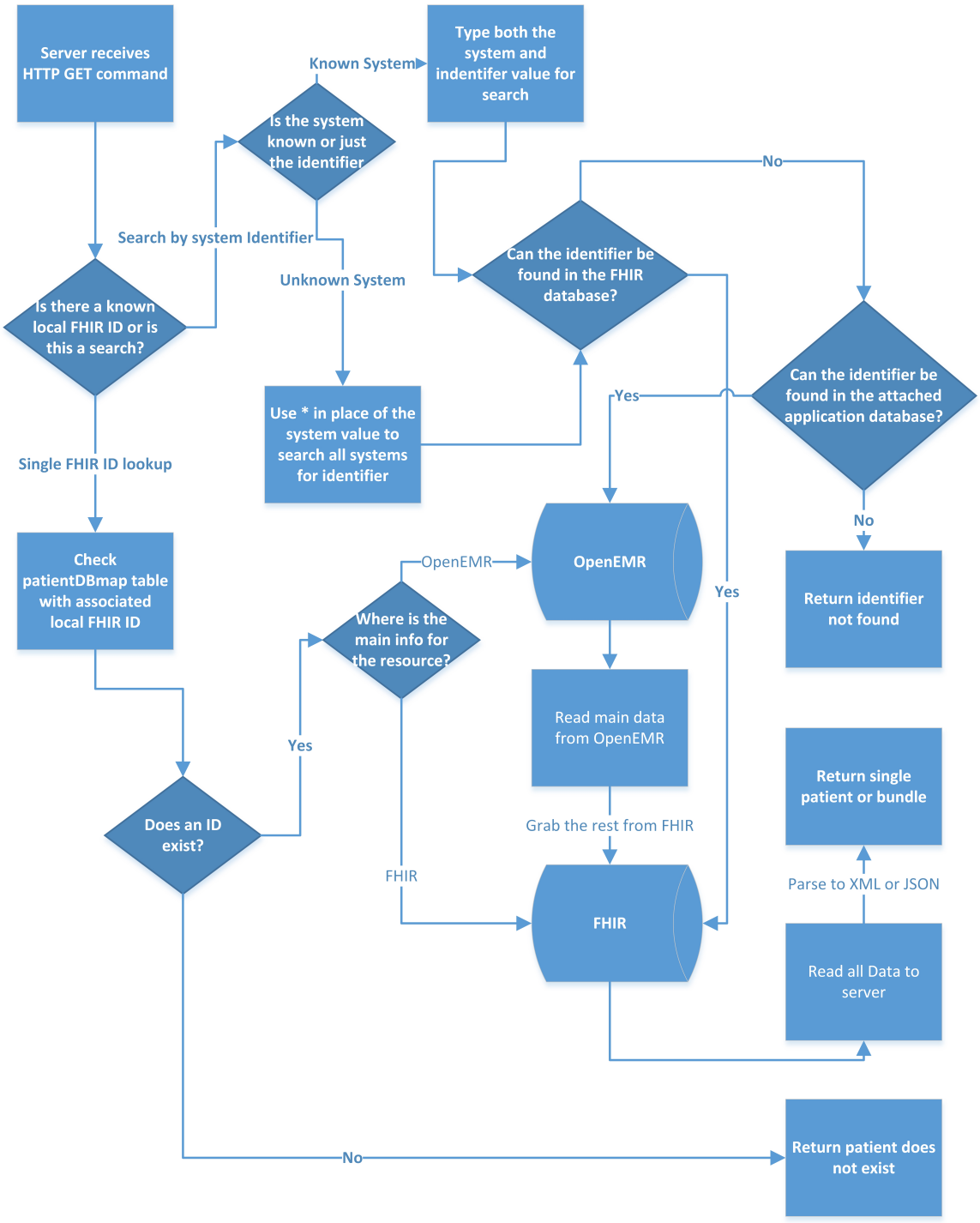

Figure 4: Logic for FHIR request of patient information

```
▼<OperationOutcome xmlns="http://hl7.org/fhir">
  ▼<issue>
     <severity value="error"/>
     <code value="processing"/>
     <diagnostics value="Resource Patient/30 is not known"/>
   </issue>
 </OperationOutcome>
```

Figure 5: `OperationOutcome` resource, showing unknown patient ID

pulled from the existing application database will be the first values in the patient resource record. When the name is pulled from the attached application database it will be marked with the value "usual" for the use value in the FHIR resource. The use value in a human name indicates the form of the name, such as a nickname or official name. According to FHIR usual indicates "Known as/conventional/the one you normally use" [4]. After the initial information is pulled from the attached application database the server will look in the FHIR database for any additional information on the patient record that is not used by the attached application.

The use of storing information in both databases allows for patients to be created and updated with any correctly formatted FHIR resource record. Even when there is more information in the record that is not used by the attached application, placing it into the other database allows for no data to be lost in the transaction of creating or updating the patient. Once the information is pulled from the corresponding database it is parsed into either a FHIR XML or JSON format, depending on which format the application or client requested.

Using a single `localID` will only return a single patient record, which in the FHIR standard is considered a patient resource. If there is not a local ID present in the HTTP GET request then the server will look at possible parameters sent along with the request. This server is currently only using the FHIR identifier token for the search parameters. This means that to do a search on this server for patient information without a local server ID, another identifier should be used and also the system value in which that identifier value is unique under. For example, if searching on the test server for a record that had an identifier of 1234 and the identifier was unique in the OpenEMR system, which for testing purposes is http://www.openemr.medinfo.com, a request for information from `http://localhost:9080/fhir-openemr /Patient?identifier=http://www.openemr.medinfo.com|1234` would be used. In the URL string, "?identifier" is used to indicate to the server that a search by an identifier token is being performed. This format for searching by an identifier token is part of the FHIR standard and will be the same on all FHIR servers.

A search can be issued through this method only if the server supports this parameter for searching. Whether or not a particular search parameter is supported by the FHIR interface is left up to the implementer of the server. It can vary from server to server. If the system value for which the identifier value is unique is unknown then two different ways can be used to perform the search. The first way would be to use the identifier value only. Using the above example, the URL would look something

like `http://localhost:9080/fhir-openemr/Patient?identifier=1234`. The second way would be to use an asterisk in place of the system value. Currently the asterisk returns the same values as the search without the system value does. The asterisk is planned to be used in future implementations of this server to search outside of the local FHIR server for patient records. The above methods will have the server search all of the available values and returns any identifiers that match, regardless of the system value, shown in Figure 6. When a search is performed a bundle resource is returned. According to the FHIR standard, "The server determines which of the set of resources it serves meet the specific criteria, and returns the results in the HTTP response as a bundle which includes the resources that are the results of the search" [5]. In the resource bundle in Figure 6, you can see that two patient resources are returned by the total value under the meta tag at the top of the XML file. Each patient resource is contained inside of it's own entry tag. Inside of each patient record there are multiple identifiers but one of the identifier values are the same between the two resources, they just have different system values.

HTTP POST

Figure 7 shows the logic when the FHIR server receives an HTTP POST command. The HTTP POST command is used to create or write information to the server database. The create process is simple, as shown in the logic diagram Figure 7.

Figure 6: Bundle returned from search interaction

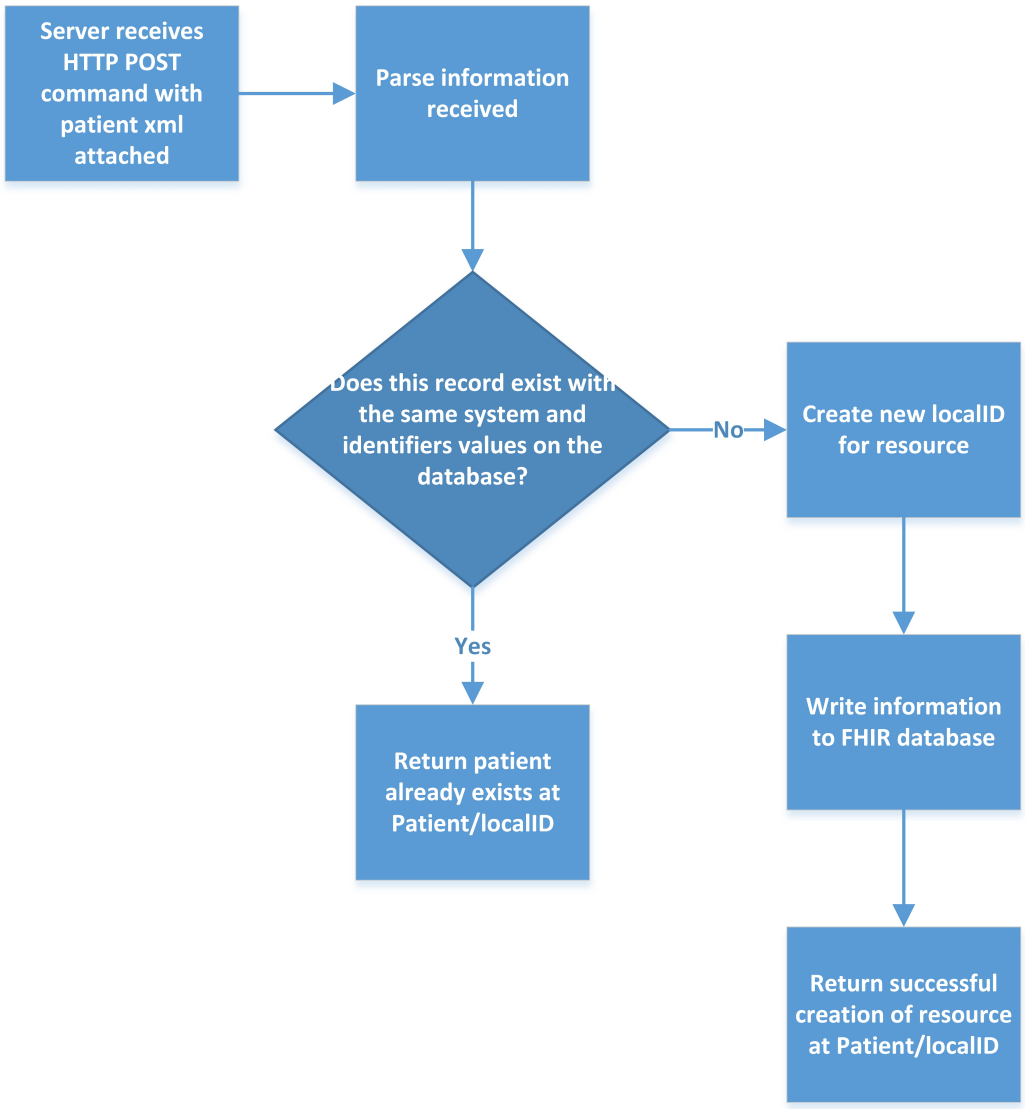

Figure 7: Create a patient resource on the FHIR server

When the FHIR server receives a HTTP POST command and a patient XML file it will parse the data and check it against the database for any identifiers and system values that are already in the database. If a match is found the record is not added to the database and an `OperationOutcome` is return to the application or client with the message "There is already a record with similar identifiers that is resource

```
C:\WINDOWS\system32>curl -i -X POST localhost:9080/Hospital-A-FHIR/Patient?_form
at=xml -H "content-Type: application/xml+fhir" --data-binary "@C:\users\vanwa_00
0\Desktop\loadFHIRpat\patientID.xml"
HTTP/1.1 100 Continue

HTTP/1.1 422 Unprocessable Entity
Server: Apache-Coyote/1.1
X-Powered-By: HAPI FHIR 1.4-SNAPSHOT RESTful Server
Content-Type: application/xml;charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 24 May 2016 21:24:10 GMT
Connection: close

<OperationOutcome xmlns="http://hl7.org/fhir">
   <issue>
      <severity value="error"/>
      <code value="processing"/>
      <diagnostics value="There is already a patient record with similar identif
iers that is resource Patient/A"/>
   </issue>
</OperationOutcome>
```

Figure 8: Failed creation of patient

Patient/ID". For testing purposes, cURL was used to send data to the server and receive a response. Figure 8 shows the HTTP POST command being issued to the server with a patient XML file and also the response from the server on a failed create.

The parameter `?_format=xml` is added to have the response from the server be in an XML format. If a `?_format=json` is used, the response would return in a JSON format even if the record being sent to the server is XML. When sending a record to the server to be created, the server needs to know what type of document it is. In Figure 8, the MIME type or "Content-Type" is set to `application/xml+fhir`. This is placed into the HTTP header when sent to the server. Also, `application/json+fhir` for the MIME type can be used. Both content types are specified by the FHIR standard.[6]

HTTP PUT

The HTTP PUT command is used to update patient information on the server. The curl command used for testing is very similar to the one used for creating a patient on the FHIR server in Figure 8. The only difference is that PUT will replace POST in the command. This indicates to the server that it is going to try and perform an update. The logic for this process is shown in Figure 9.

When the servers receives an HTTP PUT command with a resource record, it will first check to see if there is a local FHIR ID attached to the record. The update process can not occur without the local ID. If there is no ID value then the server will return a "cannot update, no identifier supplied" message. If there is an ID in the record it is compared against the `patientDBmap` table to check if there is a patient record on the system. If the patient record does not exist then the server will return a "No available resource to update with ID = `localID`" message to the user or application. If there is a record for the patient on the FHIR server then the server will next check if there is a version number or `lastUpdated` field on the resource record that has been sent to it with the HTTP PUT command. If neither of these fields are available on the new record then the server will send a "Cannot determine if newest record" message back to the application or client. If the fields exist on the received record the server will check if it is newer then the one stored in the database. If it is older

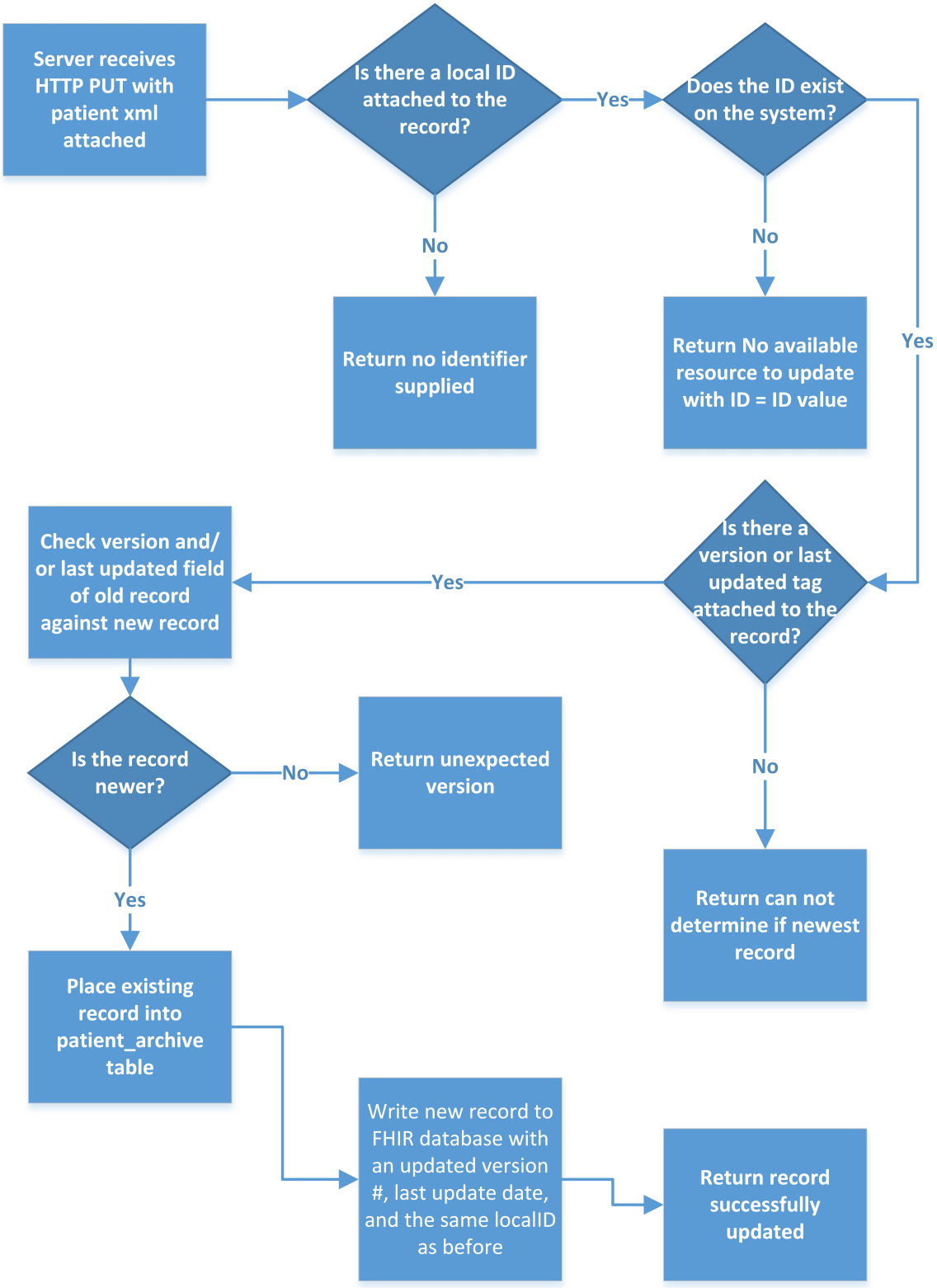

Figure 9: Updating a patient record

the server will return "Unexpected version" to the application or client. If the record is newer then the existing one then the server will place the existing record into the `patient_archive` table on the FHIR database.

The table columns for `patient_archive` have a primary key generated by the SQL server, the `localID` for the patient record on the FHIR server, the version number of the old record from the FHIR server and a resource column that consists of the old resource. This column is a `LONGBLOB` data type and is the old resource encoded in the XML format. Setting up the server to store old records will mean that data will not be lost on an update and also will allow for setting up the server to be capable of `vReads` in the future.

Attaching application database to RESTful server

In order for the RESTful server to interact correctly with the application database the server needed a way to map between the records contained in the application database and the ones contained only in the FHIR database. As mentioned earlier, the use of the `patientDBmap` table is to map the ID used for a patient record in the application database and the `localID` used for the FHIR server. All records that are stored in the FHIR database may not be used in the attached application. That means the two databases can possibly contain a different number of records and will also then

contain a different unique ID value used in each database. The `patientDBmap` will use the FHIR `localID` as a unique value for the table and will map the ID to either the applications unique ID or FHIR `localID`.

In order to setup the `patientDBmap` table, `patientInlitialize.java` was written to parse through all of the ID's of the attached application and add the them to the `patientDBmap` table. The logic for this was very simple, shown in Figure 10. When the java application is run it will lookup the ID values used on the attached application. It will then check for each ID in the attached application if it already exists in the `patientDBmap` table. If the ID is already mapped then "ID already exists" is printed to the system. If the ID does not exist in the `patientDBmap` table, then a new unique ID value is created for the FHIR database. The unique FHIR ID is added to the `localID` column, the name of the attached applications database is added to the `dbName` column and the ID value from the attached application is written to the `dbPrimary` column. The information that is stored in the OpenEMR database is mapped inside of the `attachedApplication` class of the server. The information for the FHIR patient resource is obtained from the OpenEMR `patient_data` table. The information is read into a hashmap that can be brought into the rest of the FHIR server to be read into the patient object in JAVA. The FHIR server uses this JAVA object to parse the information into either an XML or JSON format.

One difficulty when dealing with a connection to the database was dealing with the

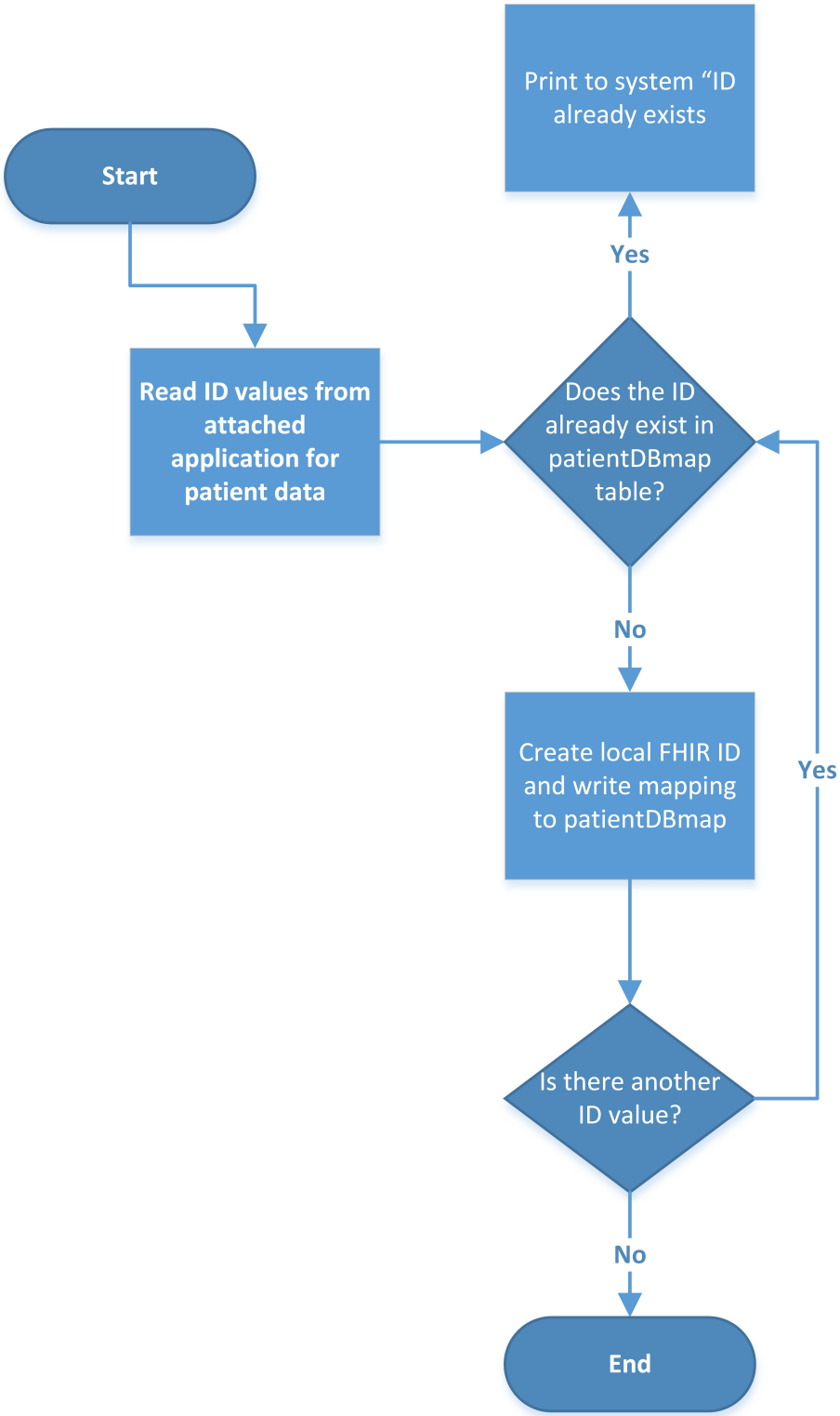

Figure 10: Setup `patientDBmap` table

deceased date value that is stored in the OpenEMR database. OpenEMR will write a "0000-00-00 00:00:00" value in the `deceased_date` column when the patient is not deceased. Their application will read this "0000-00-00 00:00:00" value as the patient is still alive. When dealing with a JDBC driver to connect to MySQL, the driver considers "0000-00-00 00:00:00" to be an illegal date value and throws an error. The work around for this was to write an SQL statement with a case statement in it. Similar to `CASE WHEN deceased_date = '0000-00-00 00:00:00' THEN null ELSE deceased_date END AS new_date`. This way a null value is passed instead of the "0000-00-00 00:00:00" date value. Any null values that are passed into the FHIR parser are ignored and not added into the record that is sent.

When adding other values into the hashmap, there are some missing values that the application database does not store as a value that can be retrieved. A simple example of this is one of the identifying values stored in the `patient_data` table in the OpenEMR database is a social security number. The value of the social security number could be read into the server and placed into the identifier fields for the patient resource, but that will leave out information that will be useful for the person or application that requested the information. Without adding in the correct code type to the identifier field the receiving application won't know what the ID value is to be used for. The code type is considered a codeable concept in the FHIR standard and for the identifier field and has some codes already defined under `https://www.hl7.org/fhir/v2/0203/index.html` [7]. These codes where added in

```
<Patient xmlns="http://hl7.org/fhir">
  <id value="6"/>
  <identifier>
    <use value="Official"/>
    <type>
      <coding>
        <system value="http://hl7.org/fhir/v2/0203"/>
        <code value="TAX"/>
        <display value="Tax ID number"/>
      </coding>
    </type>
    <system value="http://www.openemr.medinfo.com"/>
    <value value="417-02-1212"/>
  </identifier>
  <identifier>
    <use value="Official"/>
    <type>
      <coding>
        <system value="http://hl7.org/fhir/v2/0203"/>
        <code value="DL"/>
        <display value="Driver's license number"/>
      </coding>
    </type>
    <system value="http://www.openemr.medinfo.com"/>
    <value value="v12345822"/>
  </identifier>
```

Figure 11: Patient resource with identifier code type

statically to the `attachedApplication` class for the connection to the OpenEMR database. If there is a value present under the `ss` column in the patient_data table then the code of TAX and the code display of Tax ID number, will be added to the identifier type in the patient resource shown in Figure 11.

Results

The goal set in this project was to add FHIR capability to an open source EMR system. The outcome is that the FHIR server does not support all of the resources specified in FHIR as of now but the server does work with the patient resource and the same method can be applied to the rest of the FHIR resources later. It reads, searches and updates information from OpenEMR. The logic of how to deal with information about a patient from two different databases has been written in a way that will allow individuals to start the FHIR server as is or add on an application as they need it. The FHIR server can be up and running without any other interaction but the attached application data cannot be used without being setup first. The application can be added simply by applying the proper parsing of the application database to the `attachedApplications` class. With the current state of the server, the individuals setting up the server needs to have a basic understanding of JAVA, the FHIR resource structures and simple SQL commands. For future updates to this project will be the creation of a tool to automatically create the mappings between the

application database and FHIR database in order to decrease the time to add FHIR capabilities. Having the logic defined for how to read and write data between two databases will speed up the process of writing the server code for the rest of the FHIR resources. Also the logic applied in the server code can be utilized for other relational database systems and is not limited to just using MySQL or the openEMR application database. The FHIR database itself was also written using Power Architect which is a visual representation of relational databases. It was written in Power Architect [2] to simplify the creation of databases across different types of relational databases including MySQL, MSSQL, PostgreSQL, Oracle and others. With the server reading directly from the attached applications database as well from the FHIR database the information stays current and can be accessed real time. There is no need for the use of database mapping software to be running in the background to keep the databases in sync.

Discussion

At the beginning of this project there were a couple of different ideas on how to implement this system, along with a few different open source libraries that could have been used to develop with. The original idea was to have two separate databases, one database that accompanies the attached application and create another one for FHIR resources. The server would only interact with the FHIR database to store information. ETL techniques and software would then be used to move the data from the application database to the FHIR database and back again in order to update data in the application database.

There were a handful of problems with this idea. One such problem was the format of the database tables and columns that was already created for the FHIR server. Open source servers for FHIR, supporting all of the FHIR resource types, were investigated order to get the server up and going quickly. One such developed server was FHIRbase [8]. This was a server that was built to meet the FHIR standard and give the ability to store FHIR resources quickly. This type of setup may have been a full FHIR server

setup quickly but it stored its records as JSONB format in a PostgreSQL server. JSONB is implemented only in PostgreSQL databases right now and would limit the SQL management systems that could be used for this setup. In order for the information to be used with PostgreSQL and the applications database, a separate application would have to be created that would parse the information from the PostgreSQL server into MySQL or other SQL server that OpenEMR was using. Also having a separate databases and separate application would mean there would need to be a way for the information to update when changes were made. A solution would be to have the update application run on a scheduled basis. This would be something that could occur nightly. Any information that was created during the day in OpenEMR could be extracted and transformed into the FHIR database in order to update the resources. This would get a FHIR server that supports all of the resources specified in FHIR [9] up and running quickly.

A problem found with this setup was that the information was not real time. Updated information could not be queried from the FHIR server right away. The scheduled update would force the individual to wait a day or more for their information to be updated if they were trying to obtain information through the FHIR interface. The attempted solution was to create trigger events on the OpenEMR database. There would need to be an initial copy of data from the OpenEMR database into the FHIR database before using the trigger event. After the bulk data was sent, an application would be constantly running to check if data had been changed. In order

to accomplish this, a trigger event was placed onto the OpenEMR database as well as a `changedData` table was created that held an ID number and a table name that held the information to be transferred. When information was successfully created or updated in the OpenEMR database, the trigger would write the ID of the record changed and the name of the table that the information where it was changed. The application that would transfer the data to the FHIR database would continually check the changed data table for any values present. If a value is found in the table it would use the ID and table to copy the changed data to the FHIR database. After the information was copied, the `changedData` table would have the values removed and the application would return to it's original loop checking for new values. This was a near instantaneous action but it meant that I had an application that was required to be running that was separate from the FHIR server. It also meant that the same data would exist in two locations, doubling the storage space required to run OpenEMR and the FHIR server.

Using HAPI-FHIR [10] and it's JAVA libraries a server that could read the information from the OpenEMR database directly and also read from the FHIR server database was setup. The FHIR server database was also modified from storing records in a single column to containing each value in a record in it's own column in the FHIR database. This separation of values serves two purposes. First, separating the values into two different fields in the SQL database, you can write SQL queries in the FHIR server code to search for any value or parameter. Second, having the values in separate

fields will allow other applications to easily use the data stored in the FHIR server without having to use an XML or JSON parser in their code. The FHIR server was also written to read patient data from the OpenEMR database and the FHIR server database and return all information found for the patient queried. The reason that both databases store and supply the information for an individual patient is to prevent data loss while still keeping an instantaneous update of patient information from OpenEMR. If a patient record is created or updated by using the FHIR server, there is a potential for a lot of information being transmitted about a patient in the FHIR resource. The amount of information can well exceed what is used or stored by the OpenEMR database. That means that the data that is used by OpenEMR is written to the OpenEMR database and the rest of the information is written to the FHIR database. This will preserve the information that is not used by OpenEMR and prevent data loss in a patient record. Using this method still needs to have the initial setup process to determine where majority of the information is stored. Depending on the number of patient records stored in OpenEMR will determine the amount of time it takes to transfer and finish the setup. The amount of time to complete the setup continues to increase as the number of records read increase. It is believed to be a fault somewhere in the JAVA code that slows down the process the further it gets along in the reading and writing of records to the FHIR database. The more records that are read the more memory on the system was used up. This could be a problem with the JAVA garbage collection or MySQL committing of records.

Currently the process will take only ten minutes to complete if it is only reading ten thousand records from the OpenEMR database. This time increases quickly to two and a half hours when reading one hundred thousand records from OpenEMR and then slows to a crawl when reading one million records from OpenEMR. This will be investigated in future updates of this project.

Conclusion

To conclude, the server configuration was completed to correctly read information from a FHIR database and another application database. This ensures that data does not get lost when sending information from one organization to another due to the application database and the FHIR database not storing or using the same data fields. The logic created for the FHIR server can be used against other relational databases and is not limited to using just the OpenEMR database. With the current development of the server the individual setting up the server will have to write JAVA code to correctly parse their information from their application database in the `attachedApplications` class in the server. The logic used to create and attach the patient resource to the FHIR and OpenEMR database can be used to attach all other resources. This will make the rest of the implementation of a FHIR server against an application database quicker.

Bibliography

[1] http://mobihealthnews.com/news/fhir-could-be-game-changer-mhealth. *FHIR could be a game-changer for mHealth;* Stuart Hochron, MD, and Russell Leftwich, M, 2015.

[2] http://www.sqlpower.ca/page/architect. *Data Modeling & Profiling Tool: SQL Power Architect;* SQL POwer Group Inc., 2016.

[3] hl7.org/fhir/patient.html. *Resource Patient - Content;* HL7.org, 2015.

[4] hl7.org/fhir/valueset-name-use.html. *Value Set http://hl7.org/fhir/ValueSet/name-use;* HL7.org, 2015.

[5] hl7.org/fhir/search.html. *Search;* HL7.org, 2015.

[6] hl7.org/fhir/http.html. *RESTful API;* HL7.org, 2015.

[7] hl7.org/fhir/valueset-identifier-type.html. *Value Set http://hl7.org/fhir/ValueSet/identifier-type;* HL7.org, 2015.

[8] https://github.com/fhirbase/fhirbase. *FHIRbase;* health samurai, 2014.

[9] hl7.org/fhir/resourcelist.html. *FHIR Resource list;* HL7.org, 2015.

[10] http://jamesagnew.github.io/hapi-fhir/. *HAPI FHIR;* University Health Network, 2016.

[11] hl7.org/fhir/http.html#mime-type. *Content types and encdoings;* HL7.org, 2015.