Scalable Micro-Service based Approach to FHIR Server with Golang and No-SQL

Fahim Shariar Shoumik
Department of CSE
Ahsanullah University of Science and Technology (AUST)
Email: fahim.shoumik@gmail.com

Md. Ibna Masum Millat Talukder
Department of CSE, AUST

Ahmed Imtiaz Jami
Department of CSE, AUST

Neeaz Wahed Protik
Department of CSE, AUST

Md. Moinul Hoque
Associate Professor
Department of CSE
Ahsanullah University of Science and Technology
Email:moinul@aust.edu

***Abstract*—Fast Health Interoperability Resource (FHIR) for Electronic Health Record (EHR) not only opens a new era for health-care systems to exchange data between themselves but also allows other various systems following the same framework to communicate between them. FHIR is engaging and evolving rapidly and it is creating the need of massive data storage, retrieval and transfer requirements. For a server that implements this framework, has to be agile to cope with the growing and massive data handling. E-health-care data also seem to grow proportional to time. So, an ideal FHIR Server has to be scalable to fit this demand over time. Several different tools could help store, anonymize, analyze and extract data because there are no single tools to get all those things done. In this work, we propose a scalable, agile, and reliable Micro-service based Architecture for FHIR server that is highly available with the help of Document Oriented Database and helps to store health-care data. The proposed architecture is fast responsive. It gives our system the flexibility to use different tools that implement FHIR Framework interface in such way that the whole system could be made vertically scalable and the system can perform better in terms of time complexity compared to a Monolithic based implementation of the framework. Experimental results show that the proposed model can be highly used as an alternative to currently available FHIR system implementation.**

Keywords: FHIR; Interoperability; REST API; Health-Care system; scalability; micro-service

I. Introduction

FHIR Stands for Fast Health Interoperable Resource developed by HL7 [1]. FHIR can be seen as a standard that describes two major things for sharing health care records electronically among compatible systems. The first one is the data format structure for storing all sorts of medical records and the second one is the Application Programming Interface (API) on how to access those data. Medical systems implementing the FHIR standard have to preserve their data following the standard and must have interfaces to enable data sharing with other similar systems through the API. It is a framework that guides how health-care data should be stored, manipulated and searched. This framework is still under development with health-care experts around the world. FHIR framework has components for real world concepts like Patient, Practitioner, Device, Organization, Location, Health-care Service including but not limited to Allergy, Care-Plan, Observation, Diagnostics, Appointment, Medication, Task etc. FHIR has an implementation guidelines that indicates how a FHIR Server should behave over the REST API [2] Guidelines. FHIR contains a verifiable and testable syntax, a set of rules and constraints, methods and interface signatures including specifications for the implementation of a server capable of requesting and delivering FHIR business objects called 'Resource' which might be represented with JSON, XML, RDF, UML etc.

There are not too many public implementation of the FHIR standard readily available. HAPI (HL7 application programming interface; pronounced "happy") [3] is an open-source, object-oriented, Monolithic HL7 FHIR Server implementation. The project was initiated by the University Health Network, a large multi-site teaching hospital in Toronto, Canada. By default HAPI FHIR uses Apache Derby as database for its persistence layer to save resources, however it could be configured to use various database supported by JPA 2. FHIRBase[4] is another open-source relational storage for FHIR with document-API based on PostgreSQL. FHIRBase takes the best parts of Relational and Document Databases for persistence of FHIR resources. FHIRBase stores resources relationally and gives the power of SQL for querying and aggregating including a set of SQL procedures and views to persist and retrieve resources as JSON documents.
In this work, we have implemented the FHIR standard in terms of Micro-service structure rather than the monolithic based approaches. This enables our system to become agile and fast responsive and also flexible to adapt to various tools.

Micro-service [5] is a way to divide a coupled service into smaller services based on their functions which communicate with each other and gives output as a single unit. Some might consider it as Service Oriented Architecture (SOA) [6] but Micro-service is a Subset of SOA rather than SOA itself, where SOA could be monolithic as well. Micro-service gives agility in development and independent from technological stack point where each micro-service could be built with different language and database as needed but still functions like a Monolithic System architecture [7]. There is no particular rule when thinking of micro-services from monolithic, its rather application specific . To help us design the micro-service based health-care information management system, we consulted the already defined FHIR[8] components like Administration, Clinical, Financial, Security etc.

II. Reasons for Moving towards Micro-service based Architecture

Scalability

– A server is marked as scalable if the system is capable of handling the increased load. Scalability of a server nowadays is most important especially for large systems like health-care. A single monolith system is only capable of scaling through vertically[9] by increasing CPU resource for that particular system and if any portion of the system goes down then the entire application will not function. Whereas with the Micro-service architecture we have the opportunity to horizontally scale the whole system, which means we can connect more CPU that will act as a single unit but may be distributed over multiple networks. Also, we can assign more resource to a particular service handling more load and leave the others as it is.

Service Deployment

– Services should be compiled and containerize into a small container for deployment into clusters. With the help of Golang [1] Cross compilation we can compile each services into small static binaries and couple them with small Docker[10] container for deployment. Docker tools could help scaling each micro-services into large Docker Swarm Cluster which manages the internal communication network and service discovery with load balancer for the containers .

Loose Coupling

– When services are not coupled with each other for functioning and independent it becomes much more easier for developer to develop, maintain and deploy those services. On a Micro-service, a pattern follows decoupled interconnected services.

Agility and Development

– Micro-services reinforce modular structure which is highly independent, having members of all roles and skills that are required to build and maintain, which is particularly important for larger teams. Decoupling teams are as relevant as decoupling software modules.

Highly Availability and API Gateway

– Connecting to various services directly is a bad practice and if that directly connected service goes offline then the system will lose its potential to work as a single unit. To maintain a good service each micro-service should be highly available[11] with time-outs connecting to a load balancer. An API Gateway is required to manages connections to different services if one fails or responses with much delayed time.

Challenges

– Modular Micro-service pattern has some challenges, First one is decoupling the monolithic into micro-services based on business objects. After separation, an inner service communication mechanism is needed which could be through Remote Procedure Call (RPC) or a Message Bus to help exchange of messages between services. An API Gateway can help manage the client request for the whole system and forward request into separate micro-services and process the response back to the client in a meaningful format like JSON or XML.

III. PROPOSED SERVER ARCHITECTURE

Monolithic Systems like the large electronic health record systems eventually tangled into unreasonably large systems. The effect is that problems quickly get out of control thought the monolithic design is comparatively easier to debug and test if there is a need for simple changes to be made in multiple locations. The problem increases with various health care systems across different health-care ecosystems running different versions of services and not connected systems.

Breaking up a monolithic system into smaller services helps simplify the individual build for business objects. While the overall complexity of the system may increase but the savings in developer time and pain is well worth the cost. Each individual Micro-service will be much simpler to understand in isolation. In theory, this decreases the amount of time it takes to on-board new developers into a system.

A. Design Decisions

To build a Micro-service one should identify their business objects. In the health-care domain, there are no fixed sets of 'business objects', but there is a notional and ongoing evolutionary, consensus-based process for identifying some common set of business objects including terms like 'patient', 'procedure', 'observation', 'order', etc. The FHIR specification provides a framework for defining these health-care business objects known as 'Resource'. These business objects are grouped into a single category based on their similarity. For instance, Patient and Practitioner, Location, and Organization resource fall into Administration Category. Allergy, CarePlan, RiskAssessment falls into Clinical Similarly resources like Task, Appointment, Schedule fall into Workflow Category and so on. These grouped business objects have the potential to become separate services.

[1]https://golang.org

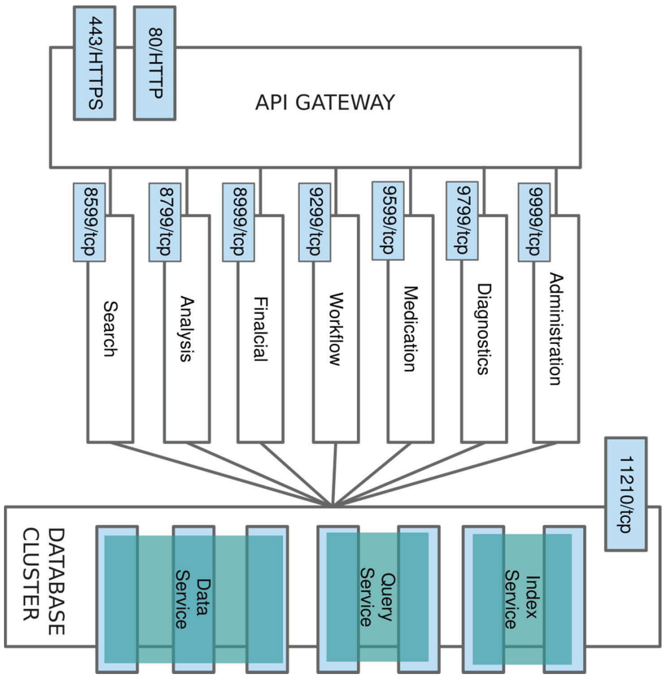

Fig. 1. Micro Service Based FHIR Server (Modularized by FHIR Components)

Figure 1 illustrates these business objects as separate micro-service in a FHIR Server for our system.

B. API Gateway

Individual micro-services are connected to the API Gateway via Googles RPC(gRPC) on different ports. The Gateway handles the routing and decides which services to talk to. It has the ability to provide the required abstractions at the gateway level for existing micro-services, e.g. rather than providing a one-size-fits-all style API, the API Gateway can expose a different API for each client.

Advantages of having an API Gateway

- Lightweight message routing at the gateway level could be used to do basic message filtering, routing, and transformation at the gateway layer.
- Central place to apply non-functional capabilities, such as authentication, security, monitoring and throttling rather implementing these services for each micro layer.

C. Independent Scaling

For our proposed model we choose Couchbase[12], which is a Document Based Storage with Multi Dimensional Scaling[13] Capability and SQL like Document Query Language called N1QL[14] which is used in FHIR Search. All those separate micro-services are connected to a Single Couchbase Database Cluster.

The database cluster in figure 1 demonstrates a deployment topology that can be achieved with Multi Dimensional Scaling Database Service. In this topology, each service is deployed to an independent zone within the cluster. Each service zone within a cluster (data, query, and index services) can now scale independently so that the best computational capacity is provided for each of them. Each micro-service could also be scaled up to multiple instances for high availability.

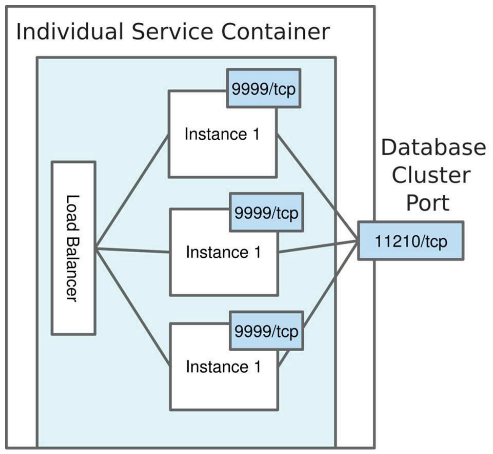

Fig. 2. Scaled Service through a load balancer (Administration)

Figure 2 shows highly available single service containers scaled up to multiple instances in a node managed through a load balancer with the help of docker. This ensures that if one instance of the service panics then the load balancer instantly switches to other instance for request processing.

D. Working Procedure of the architecture

- Communication between the client and API Gateway is done via the REST API.
- The API Gateway handles the validity of the request via Authentication and then it passes the request to the router.
- The Router decodes the request and passes the message to Service Discovery module where all the micro-services are registered. E.g, a single request might want information about a patient and diagnostic report. Those services belong to different micro-services and requests will get redirected to those service nodes via the gRPC.
- Service Nodes communicate with Database Cluster and returns the result back to the Router.
- The Router then combines the results in a Bundle Resource[15] and returns the result to the client.

IV. EXPERIMENTAL VERIFICATION

- **Experimental Setup**
  In order to evaluate the standpoint of our design, we have compared our system with monolithic based implementation with same sets data from Synthea (Synthetic Patient Population Simulator) [2]. Both of the benchmark is performed with Apache Benchmarking Tool(ab). The benchmark parameter was defined by 1000 as the total request with 100 concurrent request.

[2]https://syntheticmass.mitre.org/about.html

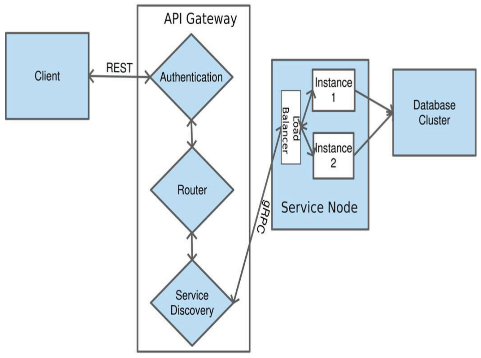

Fig. 3. Worflow Diagram of the entire system

- **The Benchmark 1**: Focuses on getting a Single Resource from both the server.
  [GET:http://localhost:8888/Patient/26d9e7b1-f633-4f85-8f1e-243455c7bc2c]

  [GET:http://localhost:80/Patient/26d9e7b1-f633-4f85-8f1e-243455c7bc2c]

- **The Benchmark 2**:
  Fetches a bundle resource that contains all the Observation results of a patient using his/her unique identifier calls [GET] were made to both server located in different port using
  [GET:http://localhost:8888/Observation?subject=Patient/26d9e7b1-f633-4f85-8f1e-243455c7bc2c]

  [GET:http://localhost:80/Observation?subject=Patient/26d9e7b1-f633-4f85-8f1e-243455c7bc2c]

Both the benchmarking are performed in a virtual machine (vagrant) with two separate instance each containing (2 logical cores) having 2 GB of Ram. Both the Monolithic and The Micro-service Server were run via docker.
Monolithic server used for benchmarking was a docker image of a HAPI FHIR Server exposing on port 8888 and assembled micro-service based FHIR Server with an API Gateway exposing to port 80. Figure 4 and Figure 5 show the results in terms of number of requests made vs response time.

In the results (figure 4 and figure 5) Monolithic system is marked with a purple colour line and our Micro-service based architecture is marked with a green coloured smooth line.

We can see from the result that, as the number of requests increases, the monolith system seem to loose performance taking longer time to response and our Micro-service based system seems to keep up with the performance over the time because the API Gateway only connects to 'Administration' service when we requested a single Patient information same thing happens for benchmark 2, it only communicates with the Diagnostics Micro-service because that is where all the models and business objects are stored for observation results.

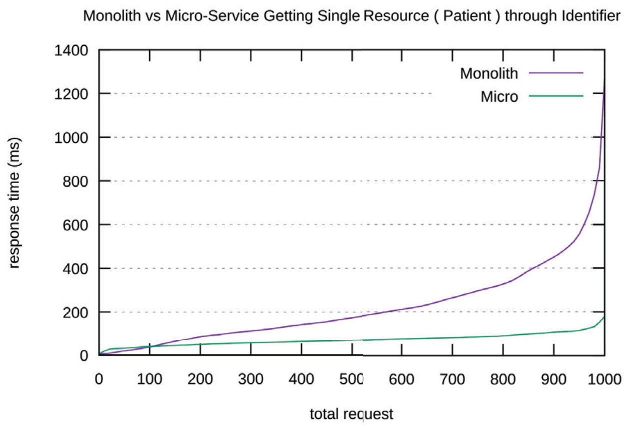

Fig. 4. Response Time for the benchmark 1 scenario

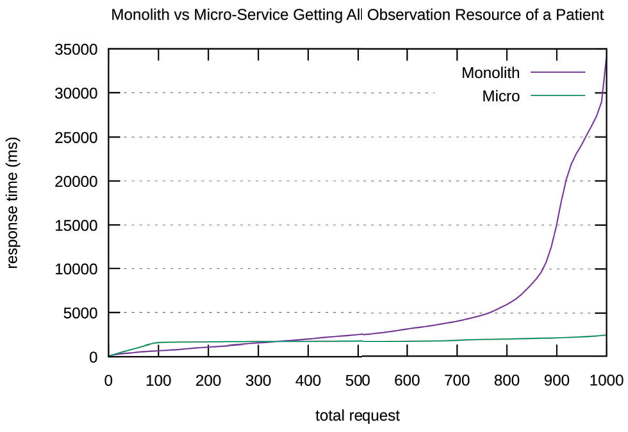

Fig. 5. Response Time for the benchmark 2 scenario

V. Discussion and Future Work

Our proposed model achieve scalability and performance by separating Gigantic Monolith based health-care application architecture into Micro-services with *Go language* and *Couchbase*. Go language helps us build a smaller static compiled binary that is easy to deploy and scale via docker and using Couch-base's Multidimensional Scaling, we can achieve performance and stability by vertically scaling the components that handle the data.

Virtualization will increase the scalability features to this proposed model. Then we can deploy each containerized micro service in different virtualization balanced through a load balancer which might have multiple cores inside it and the load balancer will choose to scale its services based on heart beat/ health check of each virtualization environment. Adding more cores may not increase the scalability but it will definitely help the load balancer to choose a better virtualization or cpu during the scaling of individual services.

This work was presented as an outcome of a continuous research work and in the next iteration of our research, we are working on other plans that include techniques to caching documents in the case of a Database failure and replication of database into multiple clusters. This will allow the system to be available in the face of known or unknown failures. Moreover, we are also working on data representation and data semantics for extraction of electronic health-care data for a complex analytical purpose which is difficult to attain in the monolithic based implementation.

VI. References

1. Joel Rodrigues. Health Information Systems: Concepts, Methodologies, Tools, and Applications, Volume 1. IGI Global, ISBN 978-1-60566-988-5, 2010

2. David Booth et al., Web Services Architecture. World Wide Web Consortium, 11 February 2004. Section 3.1.3: Relationship to the World Wide Web and REST Architectures.

3. http://hapifhir.io/doc_intro.html, Accessed on the July 16, 2017.

4. http://fhirbase.github.io/docs.html, Accessed on the July 16, 2017.

5. Richardson, Chris. ’Microservice architecture pattern’. microservices.io. Retrieved on 2017-03-19.

6. ’Chapter 1: Service Oriented Architecture (SOA)’. msdn.microsoft.com. Retrieved on 2016-09-21.

7. Rod Stephens, ’Beginning Software Engineering’, John Wiley and Sons, pp. 94. ISBN 978-1-118-96916-8, 2015

8. SMART on FHIR: a standards-based, interoperable apps platform for electronic health records.

9. Hesham El-Rewini and Mostafa Abd-El-Barr, Advanced Computer Architecture and Parallel Processing. John Wiley and Sons. p. 66. ISBN 978-0-471-47839-3, 2005.

10. Vivek Ratan, Docker: A Favourite in the DevOps World. Open Source Forum. Retrieved July 14, 2017.

11. Floyd Piedad, Michael Hawkins (2001). High Availability: Design, Techniques, and Processes. Prentice Hall. ISBN 978-0-130-96288-1.

12. https://developer.couchbase.com/documentation/server /4.6/introduction/intro.html, Accessed on the July 16 ,2017

13.Borg, I., Groenen, P., Modern Multidimensional Scaling: theory and applications (2nd ed.). New York: Springer-Verlag. pp. 207212. ISBN 0-387-94845-7, 2005

14. Andrew Slater., Ssssh! dont tell anyone but Couchbase is a serious contender: Couchbase Live Europe 2015, March 24, 2015

15. https://www.hl7.org/fhir/bundle.html, Accessed on the July 16, 2017