A FHIR-based healthcare system backend with deep cloud side security

Zoltán Szabó, Vilmos Bilicki

Abstract: The need for an easy-to-learn healthcare development framework is becoming evermore urgent, but the challenges created by the very specific nature of this domain, combined with increasingly complex technological requirements is no easy task. In this paper, we introduce our proposition for one such development stack, ready for handling large amounts of medical data and security issues. Then we introduce benchmark-based evaluations on a heavily loaded database instance, and compare them with a similar benchmark from another team, who used a similar approach with the FHIR standard but with different technologies.

Keywords: Apache Cassandra, FHIR, full-stack, security, healthcare

Introduction

The area of e-health is rapidly evolving. The focus of our developer team is not just to create a full-stack development framework that provides a simple, easy-to-learn solution for every developer who wants to enter the area of IT Health Systems, but also to make the resulting applications cloud-compatible, failure tolerant and 7/24 available. In order to establish a system like this, we not only had to face the usual difficulties of IT Health System development, such as the need for strict, yet dynamic security handling for real-life situations, but also the lack of options for some specific requirements of our system.

Our framework was written in a combination of JavaScript and TypeScript, with the server-side using the popular LoopBack framework [1] and the clients written in the Angular [2] and its extension the Ionic [3] framework. To store the healthcare data, we implemented the concept of polyglot persistence, using multiple cloud-based databases based on the areas and use cases they were designed for, in order to optimize speed, security and space requirements, while also taking the concept of cloud computing in account. Accordingly, the majority of documents are stored on an Apache Cassandra [4] cluster, while video and voice recordings from the patients and doctors are maintained in the Apache Hadoop File System [5]. We also needed a unified, officially accepted data structure, hence we chose the Fast Healthcare Interoperability Resources (FHIR) standard from the Health Level 7 (HL7) organization [6]. However, the use of the FHIR standard raised many questions for our development team, the most prominent being the issue of security. The FHIR standard provides only guidelines for implementing authorization mechanisms, such as the use of security labels as part of the meta information in the resource document and ideas to implement either the Role-Based Access Control, where the role attribute of the current user determines the enabled operations or the Attribute-Based Access Control, where the access depends on the value of an attribute in the resource [8].

For our server side, we also wanted to use the OpenAPI specification to provide our current and future developers with a simple, standardized way of creating the API endpoints. LoopBack is a readily extendable, OpenAPI-compatible Node.js framework with an emphasis on the easy creation of dynamic end-to-end REST APIs and a very supportive developer community. We used some of these community-created solutions to integrate the FHIR HAPI server [7], a popular, Java-based implementation of the standard into our stack as a database, namely the loopback-rest-connector and the loopback-sdk-builder to generate client modules for our Angular Framework codebase.

This kind of integration however leads to a rather slow backend, where the potential developers lacked full control over a key part of the data storage in the form of the HAPI server. We found that other developers either integrated the FHIR HAPI server into their developments in a way similar to our early prototype and the Smart on FHIR project [9], or went for an incomplete implementation in their own development environments like the hospitalrun.io project [10].

After this research cycle we planned a new approach, which would give us complete control over the data and the domain, is cloud-compatible, has a security module providing authorization of clinical data in real-life use cases, and even under a heavy load is at least as reliable as those in the studies we assessed during our design phase.

Our solution

Cassandra

The first step was to transfer of our data model into the Apache Cassandra database. During development, like in the hospitalrun project, we found it unnecessary to use and implement the entire set of the FHIR resources. The five components we needed to refactor for our already existing projects and solutions based on the FHIR standard were Patient, Practitioner, Observation, DetectedIssue and Location.

We also decided to use the unique indexing structure of Cassandra to optimize the partitioning and the order based on our most common queries. In our Observation model for example we chose the id for the primary key and defined the clustering order by the *effectiveDateTime* field, which records the exact timestamp when the given observation or measurement was taken, then the *code*, representing the LOINC [11] code for identifying the measurement type. These three are also the most common query parameters used in almost every query we wrote for the HAPI server when retrieving the data.

LoopBack Server

After we had succeeded in finding a good data modeling strategy, next we had to construct a LoopBack server to handle the client requests to the Cassandra database. For this, our team used the popular OpenAPI 2.0 [12] specification to describe every REST endpoint needed by our current developments. The server code was generated from this specification with the Swagger generator module, with the business logic of the endpoints later included and tested.

After completing this server, we generated the SDK modules with the loopback-sdk-builder for our TypeScript-based Angular clients; and these were successfully built into our client projects by our junior developers, replacing the vanilla HTTP calls made directly towards the FHIR HAPI server with socket-based calls towards the LoopBack server.

Security Module

The remaining issue was of course that of the security. For this domain, we designed special APM tables that described the connection among the potential applications, users and data types. A typical row in this APM database consists of the *key* field, which describes the owner of the data, the type of the data, and the application requesting the usage of the data, the *read*, *write* and *authorize* containing the identifiers of the users who are authorized to read, write or share the given resource, and finally, the *startDate* and the *endDate* timestamps signifying the valid period of the given entry. The APM database is loaded into the memory when the server is booted up.

When querying the documents, we only send the basic query parameters to the Cassandra database, such as the patient ID, the restrictions on effectiveDateTime and the code. After the LoopBack server receives a dataset, it forwards it to a filter module, which makes use of the APM records to it. This way the dataset received by the clients contains only those documents to which the authorization filter found an APM entry declaring effective authorized access.

Evaluation

After the successful implementation, next we had to benchmark our solution. When analyzing our results, we used the measurements taken from the FHIRBase project [13] for comparison, which released various benchmark results, where a PostgreSQL database held the FHIR documents. We established a test server with similar parameters to the Amazon EC2 Instance Type m3.large the FHIRBase developers used for their tests. Although our implementation had no Encounter resource, their implementation of Encounter was similar to how we used the Observation resource, in terms of the quantity of the stored information and the most common query parameters (Patient ID, Practitioner ID, effectiveDateTime, etc.)

Our results are shown in figures 1-4. While running the benchmark, the database was loaded with exactly one million Observation documents that belonged to multiple patients. The benchmark was accomplished with a lightweight Node.js application, which increased the query limit for each step,

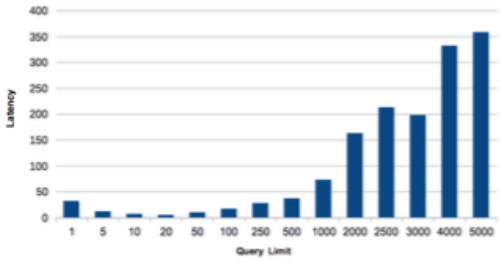

Figure 1: The query latencies when just the Patient ID is used as a filter, without any security restrictions

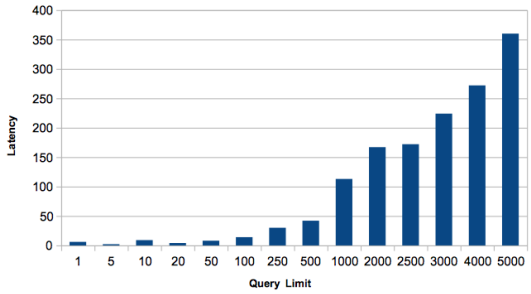

Figure 2: The query latencies when just the Patient ID is used as a filter, with active security restrictions

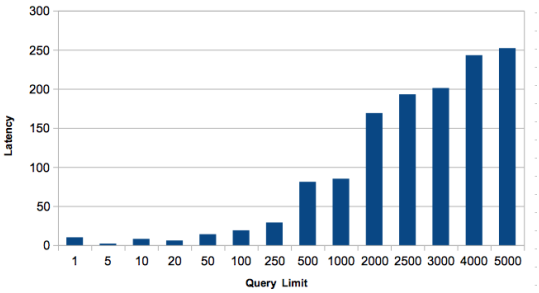

Figure 3: The query latencies when the Patient ID and the date are used as filters, without any security restrictions

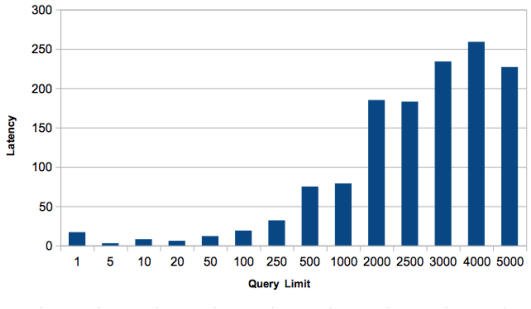

Figure 4: The query latencies when just the Patient ID and the date are used as filters, with active security restrictions

running the same select operation at first for just one document, then later for 5000. Latency is the time spent up to the server sent the response after it received the request.

These benchmarks tell us not only that despite the heavy load, our system scales at least as well as the PostgreSQL solution, or even better, but more importantly, when the security filter was active on the server there was very little difference in latency.

Conclusions

Although we still have a long road ahead of us before we have a complete, ready-to-use IT Health development stack at our disposal, it is safe to say, that our current results look most promising. Not only did we integrate a database, which is much more appropriate for the present requirements and trends of the industry than the classic, relation-based approaches, but we also created a basis for security with a fast authorization process and a design ready for multiple applications and multiple types of users. In the coming months we plan to further elaborate this framework, integrate it in our current developments, and test it with both veteran and junior developers.

Acknowledgement

This research was supported by the EU-funded Hungarian grant EFOP-3.6.1-16-2016-00008.

References

[1] *LoopBack Documentation*, Available: http://loopback.io/doc/, Accessed: 21 March 2018

[2] *Angular Architecture overview*, Available: https://angular.io/guide/architecture, Accessed: 21 March 2018

[3] *Ionic Framework Core Concepts*, Available: https://ionicframework.com/docs/intro/concepts/, Accessed: 21 March 2018

[4] Cassandra, A., *Apache cassandra*, Google Scholar, 2015.

[5] Shvachko, Konstantin, et al., *The hadoop distributed file system.*, Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010.

[6] *FHIR Overview*, Available: https://www.hl7.org/fhir/overview.html, Accessed: 21 March 2018

[7] *Introduction to HAPI FHIR*, Available: http://hapifhir.io/docindex.html, Accessed: 21 March 2018

[8] *FHIR Security and Privacy Module*, Available: http://hl7.org/fhir/secpriv-module.html, Accessed: 21 March 2018

[9] Mandel, Joshua C., et al., *SMART on FHIR: a standards-based, interoperable apps platform for electronic health records.*, Journal of the American Medical Informatics Association 23.5, 2016, pp. 899-908.

[10] *hospitalrun.io, FHIR Implementation project*, Available: https://github.com/HospitalRun/hospitalrun-server/projects/1, Accessed: 21 March 2018

[11] McDonald, Clement J., et al., *LOINC, a universal standard for identifying laboratory observations: a 5-year update.*, Clinical chemistry 49.4, 2003, pp. 624-633.

[12] *OpenAPI Specification*, Available: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md, Accessed: 21 March 2018

[13] *FHIRbase Overview*, Available: http://fhirbase.github.io/docs.html, Accessed: 21 March 2018