Building Microservices for Scalability and Availability: Step by Step, from Beginning to End

Víctor Saquicela(✉), Geovanny Campoverde, Johnny Avila, and Maria Eugenia Fajardo

University of Cuenca, Av. 12 de Abril, Cuenca, Ecuador
{victor.saquicela,geovanny.campoverde,johnny.avilam, mariaeugenia.fajardo}@ucuenca.edu.ec

Abstract. The problem of developing an application based on microservices is gaining traction over monolithic applications. Similarly to REST-based applications, their architecture may provide benefits in tasks related to their development and deployment. In this paper, we present an approach for the development and deploy of applications based on microservices using the following resources: a microservices technology software architecture, a continuous integration framework, and an environment for the deployment of microservices with high scalability and availability.

Keywords: Microservices · Continuous integration · Architecture styles · Scalability

1 Introduction

In recent years, the arriving of new requirements, new technologies, new paradigms, new methodologies, Web 2.0 applications related to the software industry, and due to some of the limitations of **monolithic** applications based on SOAP services and traditional Representational State Transfer (REST) services, microservices-based on REST have increased their presence on the Web, mainly due to their relative simplicity and their natural suitability for the Web.

However, using microservices still requires much human intervention since the majority of their software components work autonomously and contain lists of the available configurations. This makes the microservices-based software development process difficult, affecting the efficiency in the development of applications.

Traditionally, monolithic applications have focused on defining vertical architectures, and have been normally applied to SOAP and REST services and their corresponding middleware. More recently, these approaches have started to be adapted into more lightweight approaches for the development of applications based on microservices [2,5,6,17].

J. Mejia et al. (Eds.): CIMPS 2020, AISC 1297, pp. 169–184, 2021.
https://doi.org/10.1007/978-3-030-63329-5_12

In this paper, the challenge of automating the application development process using microservices is addressed by: (1) defining a microservices technology architecture that allows standardization in the use of latest generation technologies, (2) defining a continuous integration technologies framework, that allows development microservices, and (3) defining an environment for the deployment of microservices, allowing high scalability and availability of applications. The main contribution of our work is the partial automation of the process of development applications using microservices from start to finish.

The remainder of this paper is structured as follows: we first introduce some background and related work in the context of the development of applications based on microservices. Then, we describe our approach for microservice architecture, continuous integration framework, and the environment for deploy microservices. Finally, we present some conclusions and identify future lines of work.

2 Background and Related Works

In this section, we present some background information related to the Service Oriented Architecture (SOA) and specifically, to microservices. We introduce this topic by describing the current state of the art, and identifying its limitations.

A Web service is a method of communication between two electronic devices over the web (Internet). The W3C defines a Web service as a software system designed to support interoperability in machine-to-machine interaction over a network. It comprises an interface described in a machine-processable format. Other systems interact with the Web service in a manner prescribed by its description using messages, typically transmitted using HTTP with an XML serialization in conjunction with other Web-related standards[1].

Essentially, a Web service is a modular, self-describing, and a self-contained software application that is discoverable and accessible through standard interfaces over the network [4,20]. Web service technology allows for uniform access via Web standards to software components residing on various platforms and written in different programming languages.

From a technological point of view, the community distinguishes two types of Web services: classical Web services based on WSDL/SOAP (big Web services, WS-*) and Web APIs RESTful). The first have defined a stack of standards and the second are characterized for simplicity.

SOAP relies on a comprehensive stack of technology standards. SOAP plays a major role in the interoperability within and among enterprises, and serves as a basic element for the rapid development of low-cost and easy-to-compose distributed applications in heterogeneous environments [13]. APIs are characterized by their relative simplicity and their suitability for the Web. Web APIs, according to the REST paradigm [8], are commonly referred to as RESTful services. RESTful services are centered around resources, which are interconnected

[1] http://en.wikipedia.org/wiki/Webservice.

by hyperlinks and grouped into collections, whose retrieval and manipulation is enabled through a fixed set of operations commonly implemented by using HTTP. Moreover, RESTful are lighter in their technological stack.

The Representational State Transfer (REST) is an architecture style for applications within distributed environments and applies especially to Web Services. The creation of the adjective RESTful accomplished a simple ability to express whether something works accordingly to the principles of REST [8]. REST is an architecture style rather than a concrete architecture. RESTful APIs are characterized by resource-representation decoupling, so that resource content can be accessed via different formats.

The REST architectural style proposes a uniform interface, that if applied to a Web service induces desirable properties, such as performance, scalability, and modifiability, enabling Web services that facilitates working on the Web. In REST architectural style, data and functionality are considered resources. These resources are accessed using Uniform Resource Identifiers (URI), and exploited using a set of simple, well- defined operations.

The concept of microservices appears as a natural evolution of REST services within the Web. A Microservice is a small piece of code than can be deployed, executed, tested, and scaled independently [19]. The goal of a microservice is to abstract the whole functionality of a system component in a single application, making it easier to maintain and integrate with the remaining components.

Like any other application, depending on the system requirements and the development team knowledge, microservices can be developed in a wide range of programming languages and use a variety of technologies to complete the tasks for which they were created.

Regarding microservices architecture, authors of [5] apply a systematic mapping study methodology to identify, classify, and evaluate the current state of the art on architecting microservices. This study allows classifying, comparing, and evaluating architectural solutions, methods, and techniques specific for microservices.

Traditionally, monolithic systems have been developed aiming to group the functionality and its services in a single code base. Due to the simplicity of its structure, developing monolithic applications is usually less expensive than its alternatives. However, an application that concentrates all its functionality is not necessarily better, especially if it tends to grow up in complexity, users, developers and payload. Monolithic applications have enormous disadvantages compared to the use of microservices. On one hand, the implementation of new functionalities and maintainability can be as complex as extensivity of the system. A single change in the code may imply that many developers have to intervene to analyze the impact of the change and give their approval [3]. On the other hand, the use of resources can be oversized since the entire system is seen as a single module, thus being able to waste resources on components that do not necessarily require it, leaving aside components that need large amounts of resources to function properly [6]. For this reason, migrating to a microservices architecture is viable since the system is easily upgradeable and modifiable, because it is composed

of small software components that work independently. Morevoer, the facility to add resources to the services according to the needs that each one presents allows achieving better performance in both, the functionality of the system and the flow of information.

The Continuous Integration process is a highly accepted practice within the software development industry. Although there is no established framework to serve as a guide, one can find a wide variety of useful implementations that can be adapted according to the needs of each software development company. As explained in [17], the use of these practices allows teams to make continuous deliveries of products in short periods, being more productive and effective. As indicated in [21], the easiness at functionalities and changes delivery is one of the biggest advantages of applying a continuous integration process. For instance, if there is an error in a recent deployment or change, the development team receives feedback immediately so that the necessary corrective measures can be taken to update the change, and upload it again to the application server. Additionally, it is important to indicate that this process is of great help when using agile frameworks such as Scrum[2] since it allows product deliveries to be iterative and immediate, making it possible for stakeholders to observe constant progress of the project and that any change can be easily personalized [23].

The microservices architecture success corresponds directly to the adequate coordination and joint work of all the teams involved in the development, as well as in quality assurance and operations. Thus, it is necessary to use a set of good practices and recommendations that allow working with agile development methodologies, which would make it possible to quickly deliver functional software components. This concept is known as DevOps, where the terms of development and operations are combined to generate a much more flexible and robust component where all teams collaborate properly, and everything is perfectly orchestrated and automated to convey a system from development to production deployment [7,24]. Another important feature of DevOps relies on the proper use of tools that allow controlling and automating each of the stages of the software development life cycle, which in turn will allow us to generate an own framework where agile development methodologies are included, within a continuous integration process.

DevOps emerges as a movement to improve communication, collaboration, and integration between the development and IT operations teams [22]. Automation is a crucial concept for DevOps success, since every process in the software development cycle must be automated: application building from the code on the repository, automated tests, automated integration and automated deployment on each environment from development to production. From here, it is clear that continuous integration and continuous delivery are techniques that enable DevOps on any development team.

Continuous integration is a common practice in software development industry. In [16] a continuous integration process is proposed using Jenkins, using a master-slave configuration within a real-life scenario for automating test exe-

[2] https://www.scrum.org/.

cution and releasing code to production environments on multiple sites and multiple platforms. In [15], Jenkins is used for creating pipelines to completely automate continuous integration and continuous delivery process for High-Performance Computing (HPC) software. Here, every commit is automatically transformed in releasable software ensuring that all of them passes through the same validation and building process and eliminating possible errors in the circle when it is made manually. In [1], authors approach how to take advantage of Jenkins flexibility and plugins for evolving from pure continuous integration to continuous delivery. Here, Jenkins is used as orchestrator between several tools that help in CI/CD like Artifactory, chef, puppet Sonarqube and others. Finally, in [12] a pipeline from CI/CD with docker is presented. A continuous integration process produces docker images that pass through a validation process and if there are no errors, the same images are deployed by continuous delivery tools.

In this work, an efficient work model is presented, which starts from the creation of a microservice within a continuous integration process until it becomes part of a scalable and robust architecture for highly available environments. Additionally, we describe the use of open source technologies at each stage and show how they help in the automation of the entire process. Finally, it is expected that this document will be taken as a reference guide for software projects that wish to implement innovative technologies that allow the horizontal growth of their systems due to the versatility of the use of microservices.

3 Microservice Software Architecture

We present an architecture approach for developing scalable, maintainable, and robust microservices based on the Spring Boot framework and its compatible libraries, as shown in Fig. 1, to become an ordered guide for the developed of a microservice. Observe in the figure that the architecture is divided into layers. From the security to the data connection layer, all these layers will be briefly described below. In this work, we assume that there are functional and non-functional requirements acquired at some earlier stage in the software development process. Based on these requirements, the microservice will be formed of all or some of the components described in the figure.

3.1 Microservices with Spring Boot

Spring Boot[3] is a software development framework used to create microservices-based applications in Java. It also helps developers to build, deploy, and run standalone applications with a minimal effort and very simple configurations. Spring Boot brings many benefits such as autoconfiguration and a large number of compatible libraries and starter dependencies that make easy the software development cycle. Therefore, Spring Boot was selected as the standard for the development of applications based on microservices by the development team of the "University of Cuenca" in Ecuador.

[3] https://spring.io/projects/spring-boot.

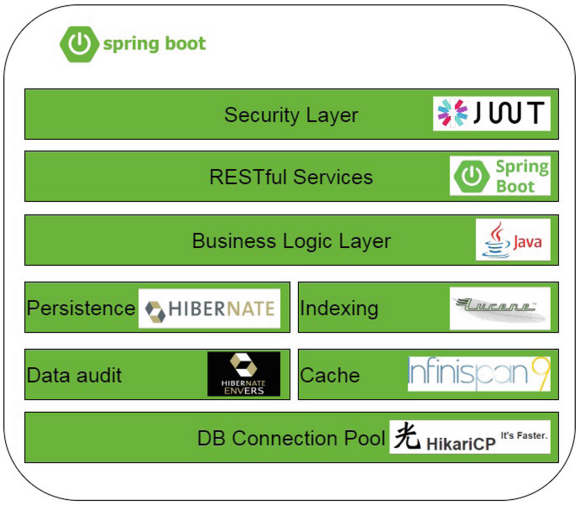

Fig. 1. Proposed microservice architecture

3.2 RESTful Services

RESTful services are widely used in the software industry due to its simplicity. REST adopts all the web precepts like its architecture and its HTTP protocol which provides functionalities for communication between system components, like well-defined actions (GET, PUT, POST, DELETE), package forwarding, cache memory and, encryption and security, each of them are important for building fast, robust, scalable and secure service-based applications.

RESTful services can be implemented easily in Spring Boot taking advantage of its auto configurable characteristics and its starter dependencies. As a matter of fact, developers will be able to build and publish RESTful services using just a few annotations in the code. Therefore, RESTful services is the architectural style used in this work for the development of applications based on microservices.

3.3 Business Logic

Business Logic is the core of each application, this layer comprises all the functionalities, which are called when a REST service is invoked. The business logic takes the data received at the REST layer, proceeds to validate and sends the results to the persistence layer for insertion, update, deletion or query objects.

3.4 Persistence

Data is the main component of any Information System, every application needs to use data stores to persist, access, or analyze information, data stores can be relational databases, NoSQL databases and others [9].

Spring (Predecessor of Spring Boot) has created various projects to create frameworks that help in the data interaction between applications and data stores in different technologies. Spring Data gather all these previous projects in a unique project whose goal is to simplify the data access over the different data store technologies, in a few related Spring Boot starter dependencies.

In the architecture proposed in this work, Spring Data JPA[4] is used for object persistence, Spring Data JPA is the implementation of JPA technology for Spring Data. JPA, widely documented in [11], is a standard for data persistence implemented by the most important data providers or ORMs, among them particularly hibernate[5], which facilitates data persistence by mapping objects in database relations (a mapped object is known as an entity). To achieve this, JPA uses well-defined annotations over the code to define entities, fields, and relations in a database. Spring Data JPA can be easily implemented in Spring Boot projects just including the Spring starter data JPA dependency on the project and annotate the code to define entities. Spring Data also provides repository classes to interact between entities and the database.

3.5 Security

Security is a crucial aspect in microservice-based applications. When compared to monolithic applications, security tends to become complex in microservices as the application grows and uses more components and services. Microservices security must be approached in two ways [18]: the first one is directly related to the number of microservices and the network monitoring that must be implemented in every service; the second one is related to internal communication between the different components of an information system, and how a vulnerability in just one component could compromise the security of the entire system. To analyze the first point, it becomes necessary to understand that although microservices break the problem down into small, easy-to-maintain parts, it granulates security and increases complexity as new components appear in the system. In large systems, security control could become a very complex task if a robust set of monitoring and control tools is not implemented. Securing a large network of microservices involves an analysis of each packet transmitted in each communication interaction, and looking for any anomalies in data transmission. Additionally, security of every microservice is also important to analyze, since the presence of a vulnerability could introduce incorrect information and compromise the integrity of the data, no only in that microservice but in the whole system. This happens since if a component sends wrong data to another, then the second component will introduce wrong data too and generate a chain reaction that damages the data of the entire information system.

To deal with security in microservices, as in the architecture shown in Fig. 1, JWT (JSON Web Token)[6] is selected as access control, security, and authorization mechanism for the communication and information exchanging between

[4] https://spring.io/projects/spring-data-jpa.
[5] https://hibernate.org/.
[6] https://jwt.io/.

the different components in a system. JWT is an open standard used in secure data ransmission, it sends information through an encrypted JSON which has a header, a payload, and a secret key. The encrypted JSON is sent as a token on the header of the HTTP request, then, the microservice decrypts the token and validates it using the same encryption algorithm and secret key that were used for the token creation. Fiunally, the microservice only executes the request whether the token is correctly validated.

3.6 Indexing

Full-text search requires the implementation of complex algorithms called search engines. Fortunately, there are free and open source search engines that can be implemented over hibernate in Spring Boot like Apache Lucene[7], Elasticsearch[8], or Solr[9].

Full-text search is achieved in basically two steps, first, data is scanned and all the words are indexed in an index store, and then, the search is executed over the indexes and not over the data. Full-text search does not only looks for exact string matching, search engines can found words that sound similar, have similar writing, words that may be synonyms, or even that result from the conjugation of verbs. The engine then sorts the results by relevance and displays them to the end-user.

To use full-text search in Spring Boot developers only have to import the correct dependencies depending on the technology of indexes what they want to use and annotate the entities and fields as indexed.

3.7 Cache

Spring Boot brings the possibility of including a cache between the microservice and the data store, this cache is used for storing the results of a function execution in an intermediate memory.

When a function with the annotation @Cacheable is executed, this technology stores the returned results and the parameters used for the execution, after that, if the function is called again with the same parameters, the microservice will return the results directly from its cache instead of executing the process.

Cache increases the performance of the microservice when it is used in the right way. However, it may involve data integrity errors if it is used without previous analysis of the methods that will use the cache, and the data retention time in this memory.

3.8 Data Audit

A data audit is directly linked with security, every change in data must be stored in history together with information about it, the old and new values, the user

[7] https://lucene.apache.org/.
[8] https://www.elastic.co/.
[9] https://lucene.apache.org/solr/.

who made the change, the time when the data was updated, the computer IP address where the change was made and other relevant information. Spring Boot uses Hibernate Envers[10] technology and annotations to achieve data audits.

3.9 Connection Pool

The last layer in the proposed architecture in Fig. 1 is the connection pool. The connection pool is a technique for sharing database connections between multiple clients for achieving better performance in data accessing and storing. The application uses it because when a request comes to the microservice, database connections have been already opened to attend that request and the application does not spend time opening and closing connections with the database.

In microservices, connection pools are mandatory due to the number of clients that could be requesting information. When a large number of clients are trying to open a connection to a database, it could result in a timeout error because the limit of permitted connections was exceeded. Connection pools deal with this problem maintaining a constant number of open connections and coordinating the requests of the clients among them. HikariCP[11] is widely used in Java applications due its simple configuration and flexibility. HikariPC reuses the database connection properties used by Spring Boot, hence, it is enough to add certain additional parameters in the configuration file and HikariCP will automatically be able to manage the connections between the microservice and the database.

4 Continuous Integration Process

The development of microservices adapts perfectly to the continuous integration cycle, due to its facility to encapsulate specific functionalities in a single module. In Fig. 2, a continuous integration framework is presented, where all the components and tools utilized are open source, which have been tailored to meet the full cycle from development to deployment of a microservice to an application server. As the Figure shows, the continuous integration cycle involves several stages, ranging from the creation of the code by the development team, through versioning tools, integration, and deployment, implementation of microservice in a server, monitoring, and reporting. In the figure, the software quality component is omitted, since there is currently a unit dedicated exclusively to the entire automatic testing and validation process.

4.1 Development Teams

Microservice development starts with cloning the initial archetype from an archetype repository to the developer machines. This initial code contains all

[10] https://hibernate.org/orm/envers/.
[11] https://github.com/brettwooldridge/HikariCP.

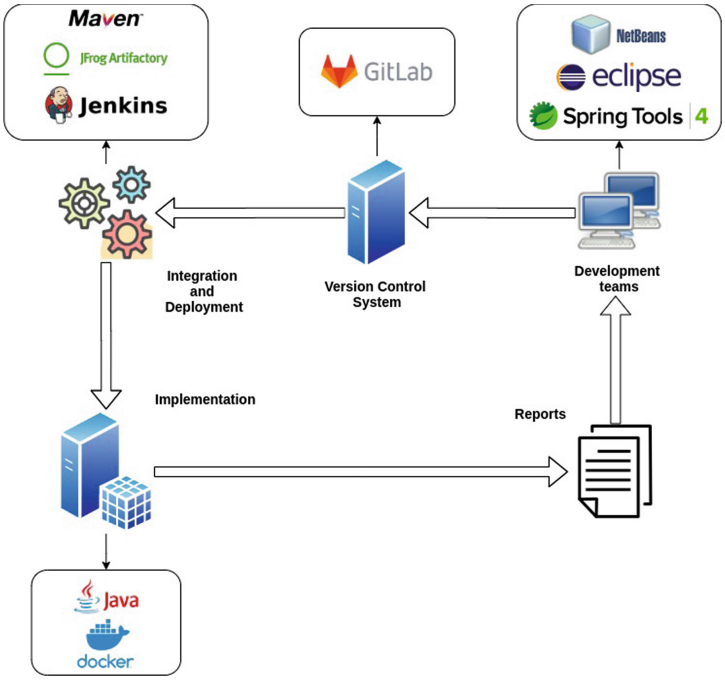

Fig. 2. Continuous integration cycle

the necessary libraries so that the members of the development team can connect their databases, external information repositories, and libraries. Eclipse[12], NetBeans[13], or Spring Tools can be used to edit the code and local tests, being Eclipse IDE the most recommended due to its adequate customization to work with SpringBoot and Java. These IDE's provide the necessary tools to perform the coding, compilation, and testing of the developed microservices before they are sent to a test or production environment.

4.2 Version Control System

A fundamental part of the continuous integration process is the implementation of a version control system, which mainly serves to keep versions of the code in an external repository and which can be accessed by all members of a development team. As explained in [10], a version control system allows you to maintain full control of all the changes that have been made in the source code of the applications. Each action carried out in the code is recorded in such a way that the author, time, and date of the changes can be known exactly. The version control system implemented in the cycle presented is Git using the GitLab tool[14],

[12] https://www.eclipse.org/.
[13] https://netbeans.org/.
[14] https://about.gitlab.com/.

which in turn acts as an information repository. Here, the code developed by the programmers is consolidated with all the changes and made it available so that it can be uploaded to an application server.

4.3 Integration and Deployment

Once the code is available in GitLab, it must be compiled to generate a functional microservice. Jenkins tool is used for this task and the ones that follow. It is in charge of automating much of the continuous integration process as explained in [5]. Jenkins is the tool in charge of connecting with the GitLab repository, obtaining the code of the branch corresponding to the environment one may want to deploy, and generating a software component called an artifact. This component is created using Maven[15], which is a tool to create compiled software packages in Java language. This artifact is stored in its artifact repository as an image. Additionally, it is important to indicate that the necessary libraries for compilation are obtained from an artifact repository called Jfrog Artifactory[16], where additionally own libraries are stored.

4.4 Implementation

The next step in the continuous integration process is to create a container with the compiled image of the microservice using Docker. To do this, Jenkins downloads the image from the artifact repository, accesses the server where the container is going to be deployed using *ssh*, and runs a command to create a container of Java with the code of developed microservice. At the end of this process, the container is deployed on the server where the command was executed and it will be available for use through the ports assigned by the server. The container with the microservice installed internally will start taking its internal configurations. In case that we want to extract the configurations so that they work in an external repository, a container volume would be necessary. The advantage of extracting the configuration is that it will be easily maintainable without the need to recompile the microservice, avoiding starting the continuous integration process again.

4.5 Reports

Reports are generated automatically by Jenkins and sent out to the operations and development team. These reports contain the result of the entire continuous integration process, including indicating whether there were any errors in the compilation or in the deployment. Reports generation is very important since it serves as feedback for both, the development and the operations teams, because, it helps improve the continuous integration process. The continuous integration process contemplates the phases from the beginning in the creation

[15] https://maven.apache.org/.
[16] https://jfrog.com/artifactory/.

of the microservice to its deployment in any environment, including, the development, testing, pre-production, or post-production. Additionally, it is important to emphasize that each stage of the continuous integration process is accompanied by the use of free software tools, flexible, and that easily integrate to automate all processes. In this way, the development teams will only upload their changes to the version control system and these will be reflected directly in an application server after the whole process has been completed.

5 Microservice Life Cycle

A microservice follows a continuous integration process to be deployed in a server application. This section explains how a microservice interacts in a real production environment where it communicates with other microservices or external services like information repositories, databases, etc. To this end, an architecture based on microservices for high availability applications has been implemented. In Fig. 3, the architecture model is presented, it corresponds to the successful implementation of architecture at the University of Cuenca, where monolithic systems were no longer used to work with microservices in a balanced and easily scalable environment. This has been possible due to the creation of a highly available environment where a load balancer, a discovery server, centralized indexing, centralized logs, and centralized cache, are all available; all these, using Docker containers within an OpenStack infrastructure.

5.1 Load Balancing

In the proposed architecture, a load balancer is implemented since in a real application, there are thousands of request that need to be efficiently handled and with a fast response. The load balancer takes all the multiple requests and send it to different instances of the microservices, and the microservices use a second internal load balancer to send request among them.

Once a request has been sent from a web application, mobile application, or another service, a load balancer called Ribbon takes such request and, using the round robbin method, sends it to an instance of the microservice. To determine which instance the request should be sent to, the load balancer uses a proxy server called Zuul[17]. This server contains the information provided by a discovery tool called Eureka[18], which describes in detail all the microservices deployed with their respective instances. If the microservice needs the information from another microservice, it uses a second load balancer with Zuul to send a request and obtain a response. This architecture does not use the same load balancer for external and internal requests, because it is necessary to have an isolated environment for the security of the internal communication of the different instances of the microservices.

[17] https://github.com/Netflix/zuul.
[18] https://github.com/Netflix/eureka.

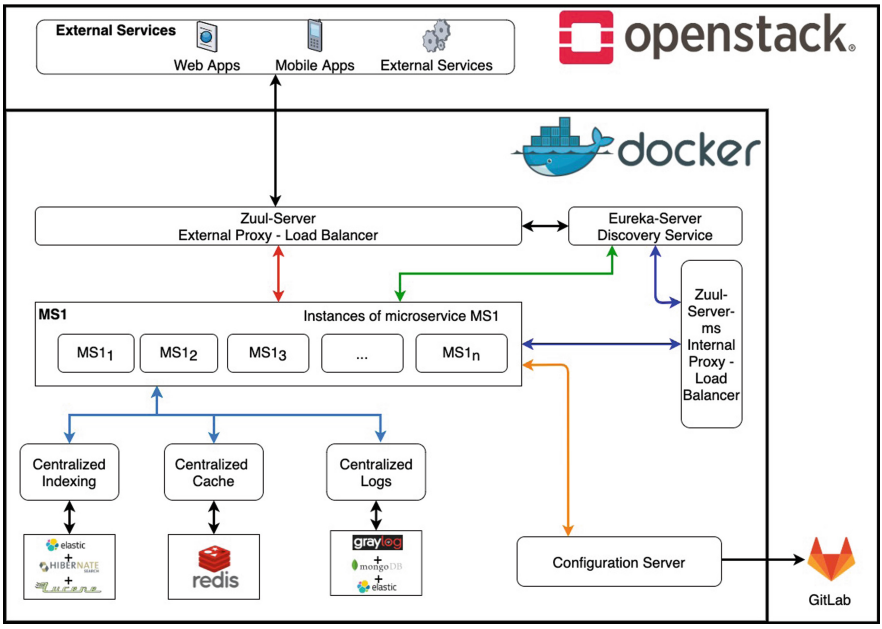

Fig. 3. Microservice life cycle

5.2 Centralized Configuration Server

As we have explained in previous sections, microservices use files to keep all the configuration isolated from the code. Nonetheless, when we have many instances from the same microservice, the scenery changes due to the difficulty to maintain as many files as instances. Fortunately, a centralized configuration server has been implemented to deal with this problem. This server, implemented with GitLab, has all the configuration files for all the microservices deployed. If a new instance of a microservice needs to be created, it takes the configuration file from the server and uses it to start. Thus, it is very easy to make any change to the configuration of a microservice, since by modifying the file corresponding to its configuration, all of these instances will inherit this new change while maintaining consistency.

5.3 Centralized Indexing

The indexes are implemented to improve responses to full-text queries. We have multiple instances of a microservice, each instance may, at a certain point in time, have different indexes because data insertions may occur for any of their instances, giving the end-user the feeling that the system has some error or that the data is inconsistent. To solve this problem in a balanced microservices environment, a centralized index system using Elasticsearch, Hibernate, and Lucene has been implemented. These tools allow each inserted data to be indexed in a

common space for all instances of a microservice, in such a way that the same response information is always available when the request goes to any instance.

5.4 Centralized Cache

Managing an independent cache in each microservice instance is just as inefficient as having indexes in each instance. To see this note that if data is updated only in the cache of the instance where the request was entered will be updated, it gives the feeling of inconsistency in the data. Eventually, all caches will have the same information since the queries will be entered by any entity and, if they are not found in the cache, they will access the database to obtain the information and leave the new data loaded in the cache. To deal with this problem, the use of a Redis[19] cluster is proposed, which is a service created and optimized to centralize the cache. The implementation of this cache cluster guarantees that all the instances of a microservice will always have the same information available, thus ensuring that the user obtains correct data and, above all, with an immediate response capacity to any update and query of records.

5.5 Centralized Logs

The administration of logs is very important in the management of computer systems as well, since, it allows developers to know in detail each of the actions that have been carried out. In a microservices architecture, we are once again tempted to deal with the problem of multiple instances that may exist. Each instance generates its respective logs, which implies that if one want to audit any microservice, one must access the logs of each instance to obtain the complete record of events. The solution to this problem is to implement a centralized log system, which allows the events of all the instances of a microservice to be registered in a single place. Thus, in the proposed architecture, the Graylog[20] tool is used, which uses Elasticsearch as its search engine on a Mongo[21] database that is responsible for storing all events. Graylog is able to recognize all the instances of the same microservice, organize their logs properly, group them and store them in the database, in such a way that the people in charge of monitoring will see a single event repository for each microservice.

6 Conclusions and Future Work

In this work, a complete cycle for the implementation of a microservices in high availability environments has been presented. The process begins with the creation of the microservice-based on an archetype and then becomes part of a continuous integration process where it will circulate through various stages

[19] https://redis.io/.
[20] https://www.graylog.org/.
[21] https://www.mongodb.com/.

until it reaches a production environment where it will be deployed in a highly available and easily scalable architecture. One of the main advantages of using a microservices architecture like the one that has been proposed, is that the systems can grow horizontally, incrementally, and quickly because the software deliveries are continuous.

The future works that are considered for the process mentioned above are the following: the implementation of exhaustive automatic quality tests to measure the overhead, performance, availability, and scalability of the architecture; the process formalization, evaluation using operational metrics and monitoring in a determined period allow improving the described process.

Acknowledgments. This work was carried out at the Direction of Information and Communication Technologies of the University of Cuenca, with the support of several people.

References

1. Armenise, V.: Continuous delivery with Jenkins: Jenkins solutions to implement continuous delivery. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering, pp. 24–27. IEEE (2015)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Software **33**(3), 42–52 (2016)
3. Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M.: From monolithic to microservices: an experience report from the banking domain. IEEE Software **35**(3), 50–55 (2018)
4. Curbera, F., Nagy, W., Weerawarana, S.: Web services: Why and how. In: Workshop on Object-Oriented Web Services-OOPSLA, vol. 2001 (2001)
5. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE (2017)
6. Djogic, E., Ribic, S., Donko, D.: Monolithic to microservices redesign of event driven integration platform. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1411–1414. IEEE (2018)
7. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Software **33**(3), 94–100 (2016)
8. Roy Thomas Fielding. Rest: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California (2000)
9. Gutierrez, F.: Pro Spring Boot. Springer (2016)
10. Hethey, J.M.: GitLab Repository Management. Packt Publishing Ltd. (2013)
11. Keith, M., Schincariol, M., Keith, J.: Pro JPA 2: Mastering the JavaTM Persistence API. Apress (2011)
12. Mironov, O.: DevOps pipeline with docker (2018)
13. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperat. Inf. Syst. **17**(2), 223–255 (2008)
14. Jiménez Quintana, J.Y.: Proposed environment for the continuous integration of performance tests

15. Sampedro, Z., Holt, A., Hauser, T.: Continuous integration and delivery for HPC: using singularity and Jenkins. In: Proceedings of the Practice and Experience on Advanced Research Computing, pp. 1–6 (2018)
16. Seth, N., Khare, R.: ACI (automated continuous integration) using Jenkins: key for successful embedded software development. In: 2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS), pp. 1–6. IEEE (2015)
17. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access **5**, 3909–3943 (2017)
18. Sun, Y., Nanda, S., Jaeger, T.: Security-as-a-service for microservices-based cloud applications. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 50–57. IEEE (2015)
19. Thönes, J.: Microservices. IEEE Software **32**(1), 116 (2015)
20. Tsalgatidou, A., Pilioura, T.: An overview of standards and related technology in web services. Distrib. Parallel Databases **12**(2–3), 135–162 (2002)
21. Vassallo, C., Palomba, F., Gall, H.C.: Continuous refactoring in CI: a preliminary study on the perceived advantages and barriers. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 564–568. IEEE (2018)
22. Waller, J., Ehmke, N.C., Hasselbring, W.: Including performance benchmarks into continuous integration to enable DevOps. ACM SIGSOFT Software Eng. Not. **40**(2), 1–4 (2015)
23. Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The impact of continuous integration on other software development practices: a large-scale empirical study. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 60–71. IEEE (2017)
24. Zhu, L., Bass, L., Champlin-Scharff, G.: DevOps and its practices. IEEE Software **33**(3), 32–34 (2016)