Available at https://jtsiskom.undip.ac.id
DOI:10.14710/jtsiskom.2022.14120

Jurnal Teknologi dan Sistem Komputer, 10(2), 2022, 60-66

Large-scale integrated infrastructure for asynchronous microservices architecture

Insan Ramadhan*), Gladhi Guarddin

Faculty of Computer Science, University of Indonesia
Kampus UI Depok, Depok, Jawa Barat 16424, Indonesia

How to Cite: I. Ramadhan and G. Guarddin, "Integrated Multiplatform Infrastructure for Asynchronous Microservices Architecture" *Jurnal Teknologi dan Sistem Komputer*, vol. 10, no. 2, pp. 60-66, 2022. DOI: 10.14710/jtsiskom.2022.14120, [Online].

Abstract - Integrated large-scale business activities increasingly rely on the use of remote resources and services across multi-platform applications. Microservice in previous research has become a solution, but this approach still leaves a data loss problem. This research methodology proposed an architecture of data transmission managed by messaging service to prevent data loss in handling many requests to deliver a multiplatform architecture, handling the plugin services, and enabling escalation based on the requirement. As a result, this research successfully implements large-scale multiplatform Single Sign-On (SSO) infrastructure for asynchronous microservices architecture. The system test results show that the developed system can handle up to 2000 requests with 20 concurrent requests.

Keywords *- SSO; CAS; OAuth; RabbitMQ; microservices*

I. INTRODUCTION

Integrated large-scale business activities increasingly rely on the use of remote resources and services across multi-platform applications. The considerable number of resources and services often requires multiple log-on, leading to credential proliferation and, potentially, security leaks. Implementation of Single Sign-On (SSO) can unify the authentication processes existing on various applications to provide centralized authentication and user data management services to support system integration [1], [2]. Large-scale infrastructure is discussed in [3], one of the performance measurements using a concurrent network through a queue, the same will be used in this study as a test parameter.

Over the past few years, microservices, also known as micro-services architecture, have emerged as the architectural style that structures an application to collect loosely coupled services to implement business capabilities. Services are modeled as isolated units, each of which can use the type of database best suited to its needs.

*) Correspondence author (Insan Ramadhan)
Email: insan.ramadhan@ui.ac.id

Compared to Service Oriented Architecture, SOA needs to share the data source with other services, causing difficulty when scaling the system. Besides, if the Enterprise Service Bus (ESB) error occurs, it will become a single point of failure that impacts the entire application. In contrast to the microservices, other services remain working, even though one of the services fails [4].

Microservices architecture was discussed in research [5], [6], such as implementing an architectural pattern, utilizing load balancers as load sharing management, and API gateway. The architecture consists of gateway and microservices codebase. Each codebase contains a load balancer and several webservers. Server scalability is resolved horizontally by increasing the number of codebase packages, and vertically by increasing hardware capacity.

Another research utilizes a web layer as an interface as well as a load balancer. It sends the request to the API gateway layer and later sends the response back to the original requester. Depending on the implementation, if there are multiple API gateways, the web layer acts as a reverse proxy and proxies the request to a specified API gateway and gets the response [7].

All those previous research focuses on load distribution using a load balancer where large-scale middleware can be scaled up as needed. This approach still has a weakness, where shortcomings occurred when loads culminated. The delivered requests may be lost due to exceeding the defined timeout. The absence of notifications causes errors on the client side. Escalating the number of codebase packages caused costs to increase.

This paper will focus on shortcomings problems when load culminated on large-scale middleware by implementing a messaging queue service to deliver an asynchronous architecture approach. Clients' requests will not be lost due to exceeding the defined timeout, and the client indeed receives response data resources from the server as expected to solve those problems.

In this paper, we describe the design and implementation of large-scale SSO and microservices architecture with message queue services, accessed through multiplatform applications such as web and mobile applications This research aims to build flexible and scalable authentication and transactional system.

Submitted: 28 May 2021; Revised: 8 February 2022; Accepted: 1 March 2022; Published: 30 April 2022

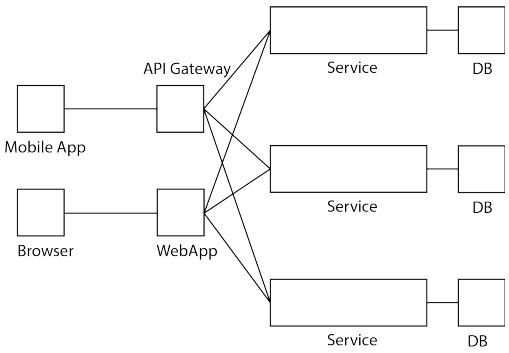

Figure 1. Microservices architecture

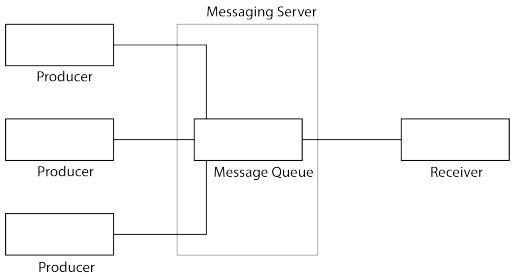

Figure 2. Message queue

The modules and applications as plugin services are appropriate for large-scale

II. METHODOLOGY

The stages carried out in this study are system analysis, system design, implementation, and testing.

A. System Analysis

The initial stage of this study is analyzed the major influence in building an integrated large-scale architecture, consisting of microservices and message queues. Microservices architecture has an advantage in developing large-scale systems such as highly maintainable and testable, loosely coupled, independently deployable, and organized around business capabilities. The architecture is suitable to solve problems in the SOA architecture which described is in Figure 1.

Figure 2 described modern cloud architecture, applications are decoupled into smaller, independent building blocks that are easier to develop, deploy and maintain. Message queues provide communication and coordination for these distributed applications. Message queues can significantly simplify the coding of decoupled applications while improving performance, reliability, and scalability.

To support the integration of multiplatform applications, we add Single Sign-On (SSO) mechanism. Single Sign-On is a technology that provides a single entry point to the corporate network while giving the user the ability to move from one portal to another without re-authentication and using portals as the preferred mechanism of interaction with the end-user

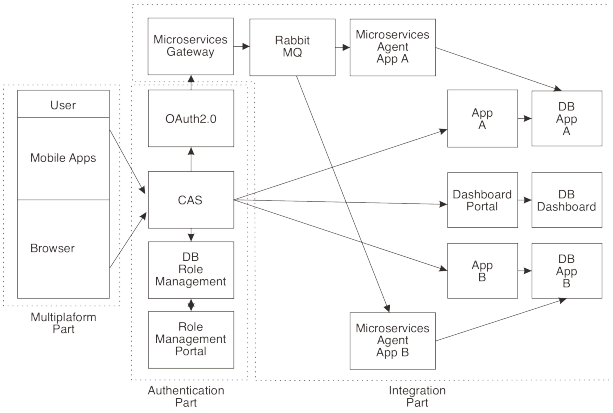

Figure 3. System architecture

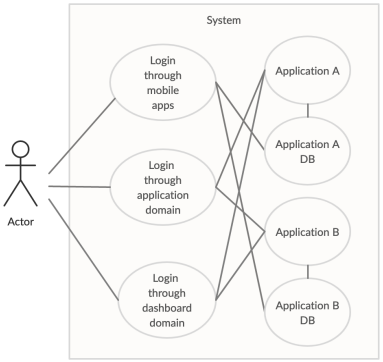

Figure 4. System use case

[8]. This research combines Single Sign-On (SSO), microservices, and message queues to deliver an architecture that can handle large-scale loads and requests for the multiplatform application.

B. System Design

The purpose of this stage is to describe the system architecture and use case, where the user can access system resources within credentials from different platforms and send a request through microservices and message queue mechanisms to handle large requests from a user. This research designed an architecture that consists of a use case, is created to access the two applications and data using the same credentials with centralized security filtering profiling. Figure 3 describes the systems architecture developed in this research.

To simulate the process, actors involved in the business process can log in through the dashboard domain or directly to the application through web and mobile applications which are described in Figure 4. Accessing the resources from a browser, an authentication process is needed. Meanwhile, the one through mobile application involves an authentication and authorization process to gain data resources.

C. Implementation and testing

From the results of the system design, the integration of multiplatform with the asynchronous process will be discussed from the multiplatform part,

authentication part, and integration part. In addition, the system will be applied in some cases.

Internal testing was conducted to ensure that the system runs as expected, where several testing scenarios were carried out against three stand-alone applications. Dashboard Application (DAP) and WordPress Application (WAP) tested on the Authentication Process; meanwhile, Mobile Application (MAP) tried on Authentication Process, Authorization Process, and Messaging Process.

III. Result and Discussion

This chapter explains the implementation of the proposed integrated infrastructure for asynchronous microservices architecture.

A. Implementation

Based on figure 3, the architecture consists of 3 main parts. The first part is a multiplatform part which consists of mobile applications and web applications. The second part is the authentication part which consists of CAS and Oauth2.0. The last part is the integration part which consists of microservices gateway, message queue, and microservices agent.

A simple mobile and web application was developed to simulate the process of user interaction. Both of the applications utilized a user interface.

This research implements CAS Single Sign-On (SSO) with OAuth on Microservices Gateway (MG) side rather than on the CAS side for the reason of making OAuth a standalone service rather than a service attached to CAS. This will facilitate further development when CAS technology is to be replaced in the future.

Various properties can be specified in CAS inside configuration files. Several CAS configuration options equally apply to several modules and features [9]. We configured the properties file as-is architecture proposed. Service registry auto-initialize from the default JSON file and set the path to JSON file location. MySql as authentication database configured in properties files either, including redirected URL connection of application URL path, query expression dialect using MySQLDialect and MySql driver class using JDBC driver. CAS can add arbitrary attributes to a registered service defined inside the CAS properties". For every new application plugged in, new services must be registered. This research workout is limited to three web applications for simulation, including dashboard application, WordPress application, and simple custom application. JSON file service needs to create, as mentioned in CAS properties. This registry inside CAS properties consists of field class set to org.apereo.cas.services.RegexRegisteredService, service set to list of application path, id is the self-defined unique number

CAS authentication model is loosely based on classic Kerberos-style authentication. An

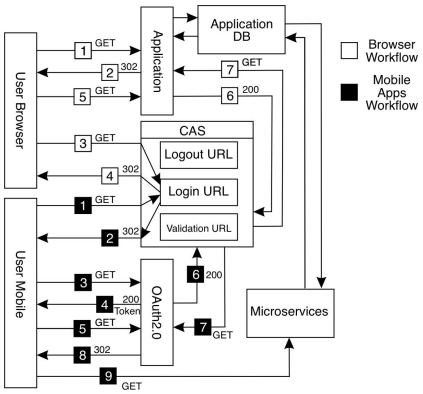

Figure 5. SSO-CAS workflow

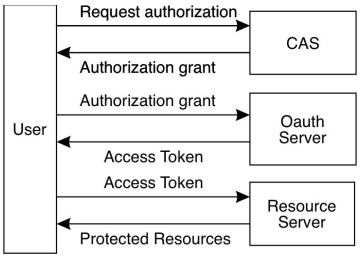

Figure 6. OAuth workflow

unauthenticated user sends a service request redirected to the authentication server (CAS Server), the authenticated user returns to the application. CAS manages certified identities and entities' passwords. CAS X.509 authentication components provide a mechanism to authenticate users who present client certificates during the SSL/TLS handshake process. X.509 is a standard format for public key certificates, digital documents that securely associate cryptographic key pairs with identities such as websites, individuals, or organizations [10]. This research requires a certified Secured Socket Layer (SSL) in its implementation.

Figure 5 describes the workflow of SSO-CAS. A client sends a request to access data resources from a browser platform; the application then checks CAS authentication status. The system will redirect the unauthenticated user to the CAS login URL, rather than to the application. CAS login session and ticket id generate and authenticate which user can access application within the session time. Application checks authentication status through validation URL every time user sends a request to the application. Unlike browser clients, mobile applications access resources through a sequential process. Starting by accessing CAS login URL to get authenticated status, Ticket ID, and session generated. Authorization request to OAuth as the second step, and the token generated as OAuth response. Token serves as a bearer in the parameter header such as ticket ID, and application ID every time request GET/ POST method is sent to microservices.

The OAuth 2.0 allows third-party applications to access protected resources in a standardized way [11]. The third party commonly hires the OAuth 2.0 protocol for authorization to access a service source [12]. It is

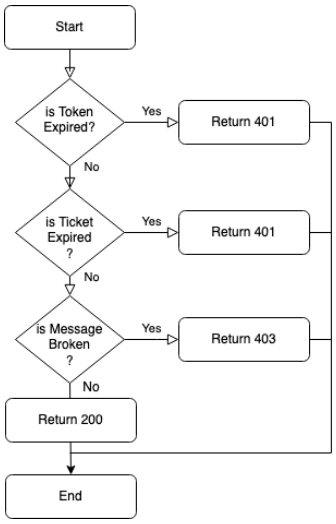

Figure 7. Microservices asynchronous workflow

most commonly implemented as a robust authorization framework in its ability and flexibly implemented on different systems and purposes as described in Figure 6.

The OAuth 2.0 protocol permits a third-party client such as an application to access a server's resources (user's profile information) with rules and permissions in a way that avoids exchanging the user's credentials [13]. This research serves OAuth as a service; a module is built, integrated with CAS by utilizing a library called phpCAS. The authorization process was conducted after authentication was successful, which was achieved by calling the CAS authentication status. Dashboard application also requires this function to enable the authentication process

The microservices architecture consists of service collection, single service independently deployed and loosely coupled by utilizing Restful API technology, reducing service complexity was made. This research uses static microservices rather than dynamic microservices, the main reason is static type microservices are fully loaded with filtered and aggregated data and these are fast in execution compared with the dynamically created microservices [14].

Figure 7 described data filtering on the microservices gateway. MG as a gate to accessing resources and security tool to prevent accessing resources when tokens and tickets have expired or the service received a broken message. Return with error response code will be delivered as an acknowledgment to user and back to the login page is a flow need to execute after.

The message queuing is an alternative to Classifications, which are complementary to the publish/subscribe model of a distributed information system [15]. Considering a large number of requests at the same time, this research placed AMQP (Advanced Message Queuing Protocol) based RabbitMQ as middleware.

RabbitMQ is the most widely deployed and popular open-source message broker. It's written in Erlang and is backed by the Pivotal Software Foundation [16].

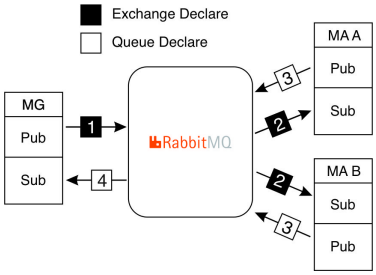

Figure 8. Microservices asynchronous workflow

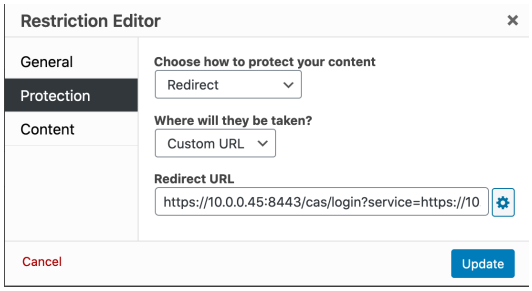

Figure 9. Content control configuration

Figure 8 describes the proposed architecture that consists of Microservices gateway (MG) and Agent (MA), where both of them act as RabbitMQ publishers and consumers in a different mode. MG is configured as exchange declare while MA is as queue declare.

Exchange Declare delivers a message to multiple consumers. This pattern is known as "publish/subscribe". The publish-subscribe (pub/sub) concept is a prevalent communication paradigm that is used across a wide range of application domains because it provides an efficient and elegant way to decouple content producers (publishers) from content consumers (subscribers) [17]. Exchanges take a message and route it into zero or more queues [15]. Microservices gateway as a publisher, a direct exchange delivers messages to queue based on application id as a message routing key. Ticket ID, Application ID, Token ID, and Request ID from the client, are put as arrays of a parameter. This explanation is already described in Figure 8 point 1.

A subscriber registers its interesting "Topic" in the form of a subscription. Messages are published to the message broker. This model allows publishers and subscribers to communicate without knowing each other's information [18]. Figure 8 point 2 described a binding relationship between an exchange and a queue. A Queue is interested in messages from this exchange. The MA message queue exchange declares mode is necessary to receive a message from the gateway service. Figure 8 points 3 and 4 described both microservices (gateway and agent) performing as publisher and subscriber at once. Service agent as publisher and gateway as subscriber set in queue declare mode. Work Queues avoid doing a resource-intensive task immediately and having to wait for it to complete. The task encapsulated a message and send it to the queue.

wp_cassify_allow_deny_order	allow, deny
wp_cassify_create_user_if_not_exist	create_user_if_not_exist
wp_cassify_enable_slo	enable_slo
wp_cassify_base_url	https://ipaddress:8443/cas/
wp_cassify_override_service_url	https://ipaddress:8443/cas/login?service=https://ipaddress/wordpress/
wp_cassify_redirect_url_after_logout	https://ipaddress:8443/cas/logout

Figure 10. WP cassify configuration

In enabling the WordPress CAS authentication, some plugins need to install. Content Control plugin is required to define protected content and CAS login URL redirection. Figure 9 shows the configuration in this research. WP Cassify plugin is essential to activate the CAS integration. This research generates custom authorization rules according to the populated CAS User Attributes. It also configures the plugin, as illustrated in Figure 10.

This research develops a cross-platform React Native Mobile Application. React Native (RN) is a cross-platform mobile application development framework acknowledged as an open-source framework by Facebook in April 2015. It combines the best parts of React Native development, a best-in-class JavaScript library for building user interfaces [19].

Application Programming Interfaces (APIs) enable communication between disparate software applications [20]. An API typically uses the REST design paradigm to manage all the ways that anyone can access. That is the best practice to integrate with CAS and OAuth through API fetching. The sequence of data sources accessed must follow the architecture determined in this research.

CAS should be authenticated before requesting the token authorization. The granted authentication process will return the ticket and user attribute. Fetching OAuth service endpoint utilizes ticket and user as the request parameters. Ticket and Token authorization resulting from the OAuth service will be used in the whole process of accessing data sources through messaging service. Figure 11 describes the flowchart of a business process in general.

Web applications and dashboards are accessed directly through CAS as a sign-on module connected to MySQL as a user management database. A portal is built separately to manage user roles. It's different from a mobile application that accesses data through synchronous and asynchronous processes after CAS authentication

We put rabbitMQ in the middle of microservices to transmit data messaging asynchronously. The OAuth2.0 and JWT token is placed to secure the data pipeline between the microservices module and database application.

B. Testing

Testing performed on a local server environment with hardware specification of MacBook Pro (15-inch, 2017), Processor 2,8 GHz Quad-Core Intel Core i7, Memory 16 GB 2133 MHz LPDDR3, Graphic Card Intel HD Graphics 630 1536 MB. Operating System macOS Catalina version 10.15.6.

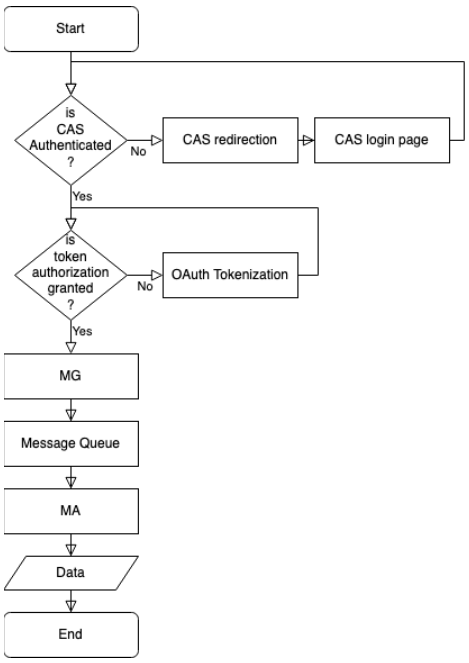

Figure 11. Mobile application flowchart

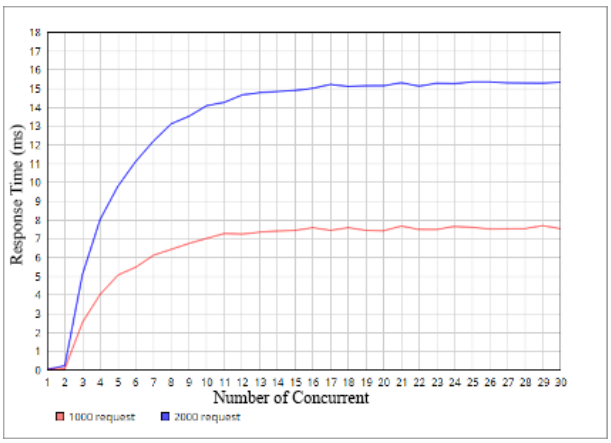

Figure 12. Load testing result

Functional tests were carried out with 8 test scenarios covering all existing modules. It starts with the CAS SSO Authentication testing on dashboard applications, WordPress applications, and mobile applications, OAuth authorization testing on mobile applications, and message delivery testing on the microservices gateway and agent modules.

The functional test results indicate that all modules are running and provide requests and responses as planned. The system is protected by the authentication and authorization process. The messaging works well, no lost requests when the service is down, and the data transmission runs automatically when the service restart, as shown in Tabel 1.

Loading tests are conducted to ensure the system can run on a large scale. The test utilizes Apache Benchmark with the number of requests between 1000 - 2000 on 1-30 concurrent requests as illustrated in Figure 12.

Tabel 1. Functional testing result

No	Test Item	Condition	Expected Result	Result
1	DAP	Not Authenticated	Redirect to CAS login URL	Passed
2	DAP	Authenticated	Dashboard URL	Passed
3	WAP	Not Authenticated	Redirect to CAS login URL	Passed
4	WAP	Authenticated	WordPress URL	Passed
5	MAP	Not Authenticated	Redirect to CAS login URL	Passed
6	MAP	Not Authorized	Redirect to CAS login URL	Passed
7	MAP	MG Down ✉ Up	Message Pending ✉ Received	Passed
8	MAP	MA Down ✉ Up	Message Pending ✉ Received	Passed
9	MAP	MG Token Not Authorized	Redirect to Login Page	Passed
10	MAP	MG Ticket Expired	Redirect to Login Page	Passed
11	MAP	MG Receive broken JSON Message	Redirect with a failure message	Passed

The test results indicate that all requests were successfully responded to without any failure, even though the response time increased, the message was kept received and processed. At concurrent 2 ~ 10, time-consuming increased significantly, above 10 ~ 15 concurrency showed decreased response time. For 15 ~ 30 showed a plateau graph that indicates able to handle more concurrency requests approximately the same result.

The result shows the architecture can handle problems without loss of transaction data when one of the services is down, in addition, the architecture developed can handle large-scale requests sent from multiplatform applications. The approach used in this research found that the system never reaches a culminated condition. Figure 12 indicates that monitoring system and scalability adjustment which is needed in previous research [5]-[7] can be solved using the proposed architecture. There was a significant increase in response time on the first iteration, but response time becomes stable after passing a certain phase even though services were down.

This research also found that the approach used in this research present a new behavior or user characteristic where the users must not expect an immediate result of their request. From an infrastructure point of view, the effect of this approach is the requester does not overload the limitations of a server and from the requester's point of view, the requester is known that the request will be processed and will get the response when it is ready.

IV. Conclusion

This research succeeded in creating a large-scale architecture through the implementation of asynchronous messaging and microservices. The functional test result shows the system can handle requests without any losing transactions when one or more services were down. This result can be a solution of Service Oriented Architecture (SOA) when a single point of failure impacts the entire application and the load test result shows that the system able to handle large-scale requests as a solution for load distribution. Even though the test results show an increase in response time but the system can handle requests without any lost transactions.

Future research is enhanced with an orchestrator module that functions to manage MG data flow to MA and vice versa. Every newly added application service is done by updating the orchestration module. Network latency and server specification are not included as test parameters in this study as a limitation.

Reference

[1] I. P. A. Pratama, L. Linawati, and N. P. Sastra, "Token-based single sign-on with JWT as information system dashboard for government," *Telkomnika (Telecommunication Computing Electronics and Control.*, vol. 16, no. 4, pp. 1745–1751, 2018. doi: 10.12928/TELKOMNIKA.v16i4.8388

[2] A. I. Ivanova and S. Vodanovich, "Single sign-on taxonomy," in *IEEE 21st International Conference on Computer Supported Cooperative Work in Design*, Wellington, New Zealand, Apr. 2017, pp. 151-155. doi: 10.1109/ CSCWD.2017.8066686

[3] S. Herrero-Lopez, J. R. Williams and A. Sanchez, "Large-Scale simulator for global data infrastructure optimization," in *2011 IEEE International Conference on Cluster Computing*, Austin, TX, USA, Sept. 2011, pp. 54-64. doi: 10.1109/CLUSTER.2011.15

[4] R. Wongsakthawom and Y. Limpiyakorn, "Development of IT helpdesk with microservices," in *8th International Conference on Electronics Information and Emergency Communication*, Beijing, China, June 2018, pp. 31–34. doi: 10.1109/ICEIEC.2018. 8473557

[5] A. Akbulut and H. G. Perros, "Software versioning with microservices through the API gateway design pattern," in *9th International Conference on Advanced Computer Information Technologies*, Ceske Budejovice, Czech Republic, June 2019, pp. 289-292. doi: 10.1109/ACITT.2019.8779952

[6] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, and S. Gil, "Evaluating the monolithic

and the microservice architecture pattern to deploy web applications in the cloud," in *10th Computing Colombian Conference*, Bogota, Colombia, Sept. 2015, pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476

[7] D. Malavalli and S. Sathappan, "Scalable microservice-based architecture for enabling DMTF profiles," in *11th International Conference on Network and Service Management*, Barcelona, Spain, Nov. 2015, pp. 428–432. doi: 10.1109/CNSM.2015 .7367395

[8] S. A. Lazarev, A. V. Demidov, V. N. Volkov, A. A. Stychuk, and D. A. Polovinkin, "Analysis of applicability of open single sign-on protocols in distributed information-computing environment," in *IEEE 10th International Conference on Application of Information and Communication Technologies*, Baku, Azerbaijan, July 2017, pp. 1-5. doi: 10.1109/ICAICT.2016. 7991757

[9] CAS Properties. (2020, Dec 10). Retrieved from https://apereo.github.io/cas/5.1.x/installation/Configuration-Properties.html

[10] C. A. Ardagna, E. Damiani, S. De Capitani di Vimercati, F. Frati, and P. Samarati, "CAS++: An open-source Single Sign-On solution for secure e-services," in *IFIP International Information Security Conference*, Karlstad, Sweden, 22-24 May 2006, vol. 201, pp. 208–220. doi: 10.1007/0-387-33406-8_18

[11] D. Hardt, *The OAuth 2.0 Authorization Framework* [online]. Available : https://tools.ietf.org/pdf/rfc6749.pdf

[12] S. R. Oh and Y. G. Kim, "Interoperable OAuth 2.0 Framework," in *International Conference on Platform Technology and Service (PlatCon)*, Jeju, Korea (South), Jan. 2019, pp. 2–6. doi: 10.1109/PlatCon.2019.8668962

[13] M. Darwish and A. Ouda, "Evaluation of an OAuth 2.0 protocol implementation for web server applications," in *International Conference and Workshop on Computing and Communication*, Vancouver, BC, Canada, Oct. 2015, pp. 2–5. doi: 10.1109/IEMCON.2015.7344461

[14] M. A. Jarwar, S. Ali, and I. Chong, "Microservices model to enhance the availability of data for buildings energy efficiency management services," *Energies*, vol. 12, no. 3, 2019. doi: 10.3390/en12030360

[15] M. Rostanski, K. Grochla, and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ," in *Federated Conference on Computer Science and Information Systems*, Warsaw, Poland, Sept. 2014, vol. 2, pp. 879–884. doi: 10.15439/2014F48

[16] Saeed Ahmad. (2020, Nov.14). *A Look at Different Open Source Message Brokers* [online]. Available: https://mrsaeeddev.medium.com/a-look-at-different-open-source-message-brokers-314862a222ac

[17] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle, "Dynamoth: a scalable pub/sub middleware for latency-constrained applications in the cloud," in *IEEE 35th International Conference on Distributed Computing Systems*, Columbus, OH, USA, Jul. 2015, pp. 486–496, 2015. doi: 10.1109/ICDCS.2015.56

[18] J. Y. Byun, Y. Kim, A. Y. Son, E. N. Huh, J. H. Hyun, and K. K. Kang, "A real-time message delivery method of publishing/subscribe model in distributed cloud environment," in *IEEE International Conference on Cybernetics and Computational Intelligence*, Phuket, Thailand, Nov. 2017, pp. 102–107. doi: 10.1109/CYBERNETICSCOM.2017. s8311692.

[19] P. Giampedraglia. (2020, June.16) *Operators of Genetic Algorithm* [online]. Available: https://www.asapdevelopers.com/best-cross-platform-frameworks/

[20] API Management. (2020, Dec 17). Retrieved from https://aws.amazon.com/id/api-gateway/api-management