

Identity distribution and session handling in microservice architecture

S.U Kumarasingha

2017



Identity distribution and session handling in microservice architecture

**A dissertation submitted for the Degree of Master of
Science in Information Security**

S.U Kumarasingha

University of Colombo School of Computing

2017



Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: Sameera Udeshika Kumarasingha

Signature

Date

This is to certify that this thesis is based on the work of Mr. Sameera Udeshika Kumaraisngha under my supervision.

The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr.D.A.S. Atukorale

Signature

Date

Abstract

Microservices are getting increasingly popular in past couple of years because of their distinct advantages over monolithic applications. In monolithic systems all the services are running in a single same machine or container, but in microservices these services are running in separate containers and they collectively working to form a larger application. This research addresses two basic problems in such kind of system, first problem is how to authenticate users and distribute identity information in a microservice system. And the second problem is how to create and handle user sessions in a microservice system. Both of the first and second chapters of this dissertation gives a solid overview about the research and background of the problem, including a detailed description about microservice architecture and various authentication and authorization mechanisms.

In this study, external access delegation system been used to handle user authentication. Then two back-ends developed to distribute and handle the identity information among microservices. Basically in first case, user authentication process handles by each of the microservices and in the second case user authentication process handle by a centralized reverse proxy. Then to handle sessions, a token will create with necessary information about the user session and it will distribute among internal microservices. Both of the chapter 3 and 4 in this dissertation concerned to design and implement this solution. Additionally, these chapters describes security and performance issues which occurred during the design phase of the system and various strategies that has taken to overcome them.

Finally, various end to end and component level tests performed to measure the performance and security tests were made based on OWSAP top ten vulnerabilities. Chapter 5 in this dissertation contains detailed information about these tests and their results. On overall system shows good efficiency in both of security and performance. The user authentication process handled using a central reverse proxy showed much less performance, but it has several advantages over the process which authenticate users at microservice level. Final chapter in the dissertation contains a detailed description about this conclusion and how the test results lead to made this conclusion.

Acknowledgements

In last two years, I studied information security in university of Colombo. The knowledge I gained in this period helped me a lot to do this research. I would like to thank all of the UCSC lectures who help me to improve my information security knowledge. And most importantly, I am very much thankful to the Dr. Ajantha Athukorala who helped me a lot as the supervisor and showed me the right direction to gain success in this research. Other than that, I am thankful to Sam Newman who is author of the book “Building microservice”, this book helped me a lot in gain basic knowledge about microservice. Finally, I would like to thank everyone who helped me to gain success in this project.

Table of Contents

Declaration.....	ii
Abstract.....	iii
Acknowledgements	iv
Table of Contents	v
List of Figures.....	viii
List of Tables	ix
List of Abbreviations	x
Chapter 1: Introduction	1
1.1 Research problem.....	1
1.2 Scope and Objectives	2
1.3 Deliverables.....	2
1.4 Content of each chapter.....	3
1.5 Summary of chapter	3
Chapter 2: Background	4
2.1 Key Benefits of microservices over monolithic systems	5
2.2 Service oriented architecture vs Microservice architecture	6
2.3 Security in Microservices.....	6
2.4 Authentication and Authorization in Microservices	8
2.5 OAuth 2.0.....	8
2.6 OpenID.....	10
2.7 JSON Web Tokens (JWT)	11
2.8 Token based authentication and session handling	12
2.9 Kerberos	14
2.10 Securing service to service communication.....	15
2.11 OWSAP security standards for system vulnerabilities.....	16
Chapter 3: Design of Solution.....	18
3.1 Authentication Server.....	18
3.2 Architecture.....	19

3.2.1 Back-end 1 – Authenticate users by microservices	19
3.2.2 Back-end 2 – Authenticate users by a reverse proxy	20
3.4 Caching database.....	22
3.5 Reverse proxy to service communication	23
3.6 Handling user sessions	24
3.7 Terminating user sessions (User logout)	25
3.8 Securing the system	27
3.8.1 Secure Communication with HTTPS.....	27
3.8.2 Token Validation	27
3.9 Summary of chapter	27
Chapter 4: Implementation.....	29
4.1 Backend services	29
4.2 Authentication Service	31
4.3 Implementation of first backend	31
4.4 Implementation of second backend.....	32
4.4.1 Reverse Proxy	33
4.5 Data Caching	34
4.6 Database Usage	34
4.7 Implementation of reverse proxy to microservice communication.....	35
4.8 Implementation of logout service.....	35
4.9 Client Implementation	36
4.10 Technologies used	38
4.11 Summary of Chapter	39
Chapter 5: Results & Evaluation.....	40
5.1 Performance Benchmark Results	40
5.1.1 End to end level performance evaluation.....	41
5.1.2 Component level performance evaluation	43
5.1.3 Performance Evaluation.....	44
5.2 Security test results	44
5.2.1 Dos Attacks.....	44

5.2.2 Reverse Proxy to Service Communication	45
5.2.3 Security result evaluation	45
5.2.4 Security evaluation against OWSAP top ten vulnerabilities.....	46
5.3 Functionality evaluation.....	47
5.4 Overall system evaluation.....	48
5.5 Summary of chapter	48
Chapter 6: Conclusion & Future Work	49
6.1 Future work	50
References	52

List of Figures

Figure 1 Monolithic authentication and session handling	2
Figure 2 Microservices equivalent authentication and session handling	2
Figure 3 Monolithic Architecture	4
Figure 4 Monolithic architecture – Service stack	4
Figure 5 Microservice Architecture and a microservice	5
Figure 6 Stack of a microservice	5
Figure 7 OAuth user authentication.....	9
Figure 8 Cookies based user authentication	13
Figure 9 Token based user authentication	14
Figure 10 Back-end 1 Authenticating users in microservices	19
Figure 11 Back-end 1 Authenticating users in microservices - Sequence diagram	20
Figure 12 Back-end 2 Authenticating users in microservices	21
Figure 13 Back-end 2 Authenticating users in microservices - Sequence diagram.....	22
Figure 14 Reverse proxy to microservice communication.....	24
Figure 15 Architecture for user logout	26
Figure 16 Implementation of Back-End 1	32
Figure 17 Implementation of Back-End 2.....	33
Figure 18 Implementation of logout mechanism.....	35
Figure 19 Command line client	36
Figure 20 GUI client user login.....	37
Figure 21 GUI Client Main Window.....	37
Figure 22 Performance evaluation user authentication back-end 1	41
Figure 23 Performance evaluation user authentication back-end 2	42
Figure 24 Performance evaluation for end to end request and response back-end 1	42
Figure 25 Performance evaluation for end to end request and response back-end 2.....	43
Figure 26 User authentication in reverse proxy.....	43
Figure 27 Time for user requests in microservice	44
Figure 28 Time for user request with dos attack	45

List of Tables

Table 1 Key benefits of microservices over monolithic systems	6
Table 2 Security issues in microservices	7
Table 3 Authentication and authorization mechanisms.....	8
Table 4 Roles in OAuth authentication process	9
Table 5 Security threats in OAuth protocol.....	10
Table 6 JWT claims	12
Table 7 Problems with cookie based authentication	13
Table 8 Advantages of token based authentication	14
Table 9 Advantages and disadvantages of Kerberos protocol.....	15
Table 10 OWSAP top 10 vulnerabilities	17
Table 11 Securing service to service communication	23
Table 12 Services in the backend	29
Table 13 Example operations and rest urls.....	30
Table 14 Services in the backend	31
Table 15 Technologies used in this system	38
Table 16 System specification.....	41
Table 17 Results of security tests	45
Table 18 Broken authentication and session management tests.....	46
Table 19 Functionality evaluation	48

List of Abbreviations

CSS	Cascading Style Sheets
DOS	Denial Of Service
GUI	Graphical User Interface
HMAC	Hash Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IP	Internet Protocol
JSON	JavaScript Object Notation
JWE	JSON Web Encryption
JWT	JSON Web Token
KDC	Key Distribution Center
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MAC	Message Authentication Code
OS	Operating System
PKI	Public Key Infrastructure
RPC	Remote Procedure Call
SAML	Security Assertion Markup Language
SOA	Service Oriented Architecture
SQL	Structured Query Language
TGT	Ticket Granting Ticket
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
XSS	Cross Site scripting

Chapter 1: Introduction

Traditionally most of the computer systems built using monolithic architecture. In monolithic systems all services or the entire implementation is running in a single machine. This kind of monolithic systems have used for many years. But with the introduction of cloud services, multiple clients and with high network capacities monolithic systems are hard to maintain and less practical. Because of these reasons microservice architecture introduced as an efficient alternative to this kind of systems.

In microservice architecture, services are running in separate machines or containers. Because of this containerized nature, services can't communicate and pass data between each other internally or locally. So microservices are using REST, SOAP or XML based protocols to communicate with each other. With these new system changes, most of the security practices used in monolithic systems become obsolete. Now system designers have to find new ways to secure service to service communication, secure ways for user authentication, authorization, session handling and etc.

Authentication is a core security concept of any kind of system. In authentication process, the system confirms that a party is who he says he is. The first problem this research address is, how to authenticate users in the microservice systems and distribute user information among services. There are various ways to handle user authentication from local servers to access delegation services. In this research user authentication performed using an access delegation services and various combinations of tokens used inside and outside of the perimeter to secure the authentication process.

Basically, two back-end architectures developed to perform user authentication. In first back-end, user authentication performed by individual microservices and in the second back-end user authentication performed by a reverse proxy server. Various measures have taken to ensure security and performance of this system. A separate caching database used to improve performance and some services used as asynchronous services to reduce overhead to the system. Different token types used inside and outside of the system, token validation mechanisms used to prevent replay attacks and requests with malformed token. All the data passing among microservices signed using public/private key pair to avoid man in the middle attacks.

The next topic addressed by this research is user session management. User sessions are used to represent authenticated users. This system used a token based session mechanism, rather than using cookie based session mechanisms and it has various advantages over the other. The user session should destroy on the event of user logout, in this system a publisher/subscriber event model used to inform other microservices about the event. Finally, both of the back-ends implemented on Docker and several client applications implemented to test the system behavior. Furthermore, various tests including capacity tests, end to end level tests and component level tests were performed to measure the efficiency and security of the system. Then the system tested against the top 10 vulnerabilities listed in OWASP standards.

1.1 Research problem

User authentication is the process of identifying and verifying the user who he says he is. We can summarize the user authentication process in monolithic systems as shown in Figure 1, this system will first figure out who the caller is and then the user credentials will pass to other services and

user session data will store in a repository for later usages. If we try to use the same technique for microservices, then we will have to implement independent security barriers at each of the microservices as shown in figure 2. This architecture is inefficient because of several reasons. If we need to change the authentication mechanism, we will have to make the same change in each of these authentication barrier. Furthermore, if we are using different technologies in each of the microservice, then we will have to implement the same functionality in different technologies. So apparently we can't map the same authentication process in monolithic systems to microservice system. The first problem which address by this research is to design a secure and efficient architecture to manage and distribute user identities among services.

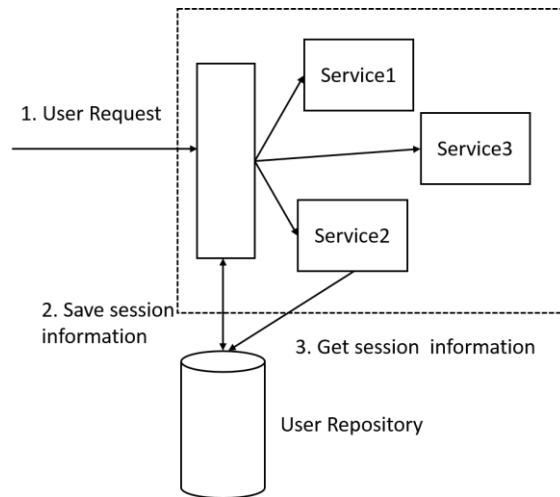


Figure 1 Monolithic authentication and session handling

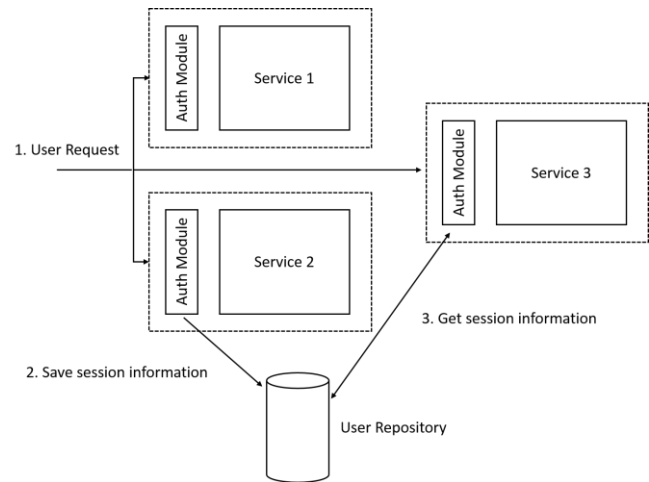


Figure 2 Microservices equivalent authentication and session handling

As in monolithic systems, microservice systems have to create user sessions. There are various ways to create user sessions, such as creating server side sessions using databases such as Redis or token based session creation using OAuth services and web tokens. Each of these have different advantages and disadvantages. In here I will design the protocol to find an efficient way to create user sessions, destroying user sessions and sharing them among other microservices. This research address above two problems with the focus of improving security and performance.

1.2 Scope and Objectives

Objectives of this research is to find a better solution for user authentication and to distribute authentication details among microservices. The authentication details distributing process should be performance efficient and secure. To evaluate this, multiple microservice architectures will be implemented and those will be compared. Furthermore, the research aims to find an efficient and secure solution to handle user sessions and destroying user sessions in microservice based systems. Then the overall security of the system will be measured against standard security practices. Finally, it is expecting to use standard protocols as much as possible.

1.3 Deliverables

A number of deliverables will be implemented to demonstrate the success of this project. Two backend systems will design based on Docker. In first backend microservices will perform the

authentication and in the second backend authentication process will handle by a reverse proxy. In these two back-ends couple of microservices will implemented using different programming languages. A command line based client and a desktop based GUI client will implement to test the microservice system. Furthermore, test results will obtain for end to end and compoment level capacity and security tests.

1.4 Content of each chapter

Chapter 1: Introduction – This chapter contains an overview of the complete system, objectives of doing this research and a detailed description about the research problem.

Chapter 2: Background and similar systems– This chapter include information that the reader may not already familiar but required to understand the rest of the project. Other than that, systems similar to the system designed in this research will discussed.

Chapter 3: Design of Solution – This chapter contains top-level design details about the system, including various measures taken to improve the system performance and security.

Chapter 4: Implementation – This chapter describes the system at a finer level of detail, including various technologies used to implement the design made in chapter 3.

Chapter 5: Results & Evaluation – This chapter contains results of the performance and security evaluation of the system. Furthermore, a discussion about advantages and disadvantages of the system is included at the end.

Chapter 6: Conclusion & Future Work – This chapter contains a summary of the project, effectiveness and limitations of the system. Future work section contains things that can do to improve system more.

1.5 Summary of chapter

This chapter contains a brief background and a high level overview about the entire system. The research address problem with user authentication and session handling in microservices. Here the objectives are to improve performance and security in such kind of system. The system will be implemented using two architectures, then the security and performance of these systems will be compared. Furthermore, a Docker based microservice system, a command line client and a GUI based client will be implemented as deliverables of this research.

Chapter 2: Background

Traditionally all system components were built as a single system, this kind of systems are known as monolithic systems. In which, multiple services or a set of services are running in the same environment and same operating system. Sometimes these services may replicate to increase performance, but still entire system will depend on a different copy of the same environment. Following figure 3 shows the architecture of such a system. This kind of systems are easy to use, all the services can build together and services can communicate between each other locally inside the boundary of the system. But developers start to face many kind of problems when starting to deploying this kind of system into cloud environments. Especially, these systems make it hard to handle service deploying, scaling and implementing services with multiple technologies. As shown in the figure 4, monolithic systems may contain multiple services in a single machine.

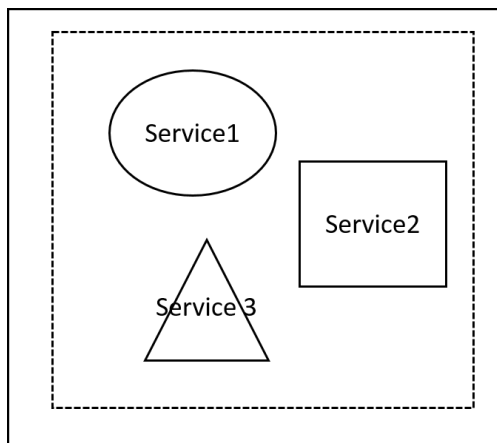


Figure 3 Monolithic Architecture

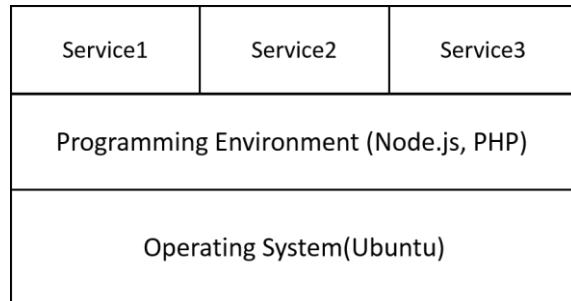


Figure 4 Monolithic architecture – Service stack

But in microservices architecture the services are further apart and each of the services isolated in its own system as shown in figure 5. As shown in figure 6 a service will run in its own programming environment and in its own operating system. We can think about the system as a set of small, autonomous services working together. One of the most important property in this kind of environment is the ability to implement services using different technologies. Unlike monolithic systems, microservices can no longer communicate with each other locally, instead of that they have to communicate with each other using a REST, RPC or any other similar protocol.

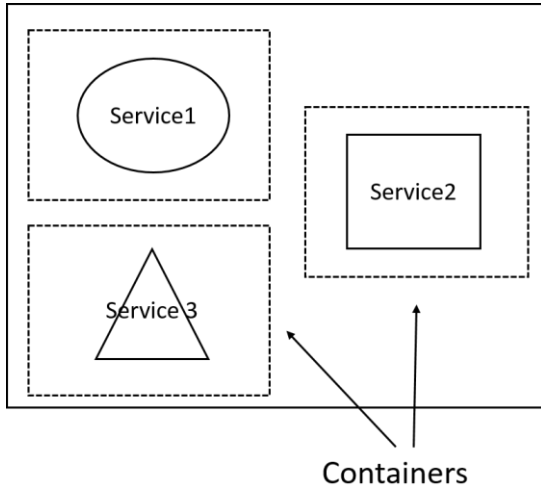


Figure 5 Microservice Architecture and a microservice

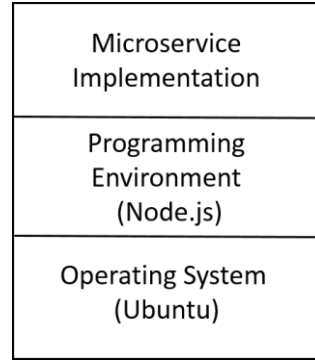


Figure 6 Stack of a microservice

2.1 Key Benefits of microservices over monolithic systems

Table 1 lists key benefits of microservices over monolithic systems.

Benefit	Description
Technology Heterogeneity	In monolithic systems every services have to program in same language or we will have to run multiple programming environments in the same environment. This doesn't allow us to use the most suitable language for each of the component and if developer has a need to change the programming language of the components, they will have to change all the components at once. But with microservices each microservice can program with best suitable technology, because they are running in seperate containers.
Resilience	In monolithic systems, if one service fails, entire system will fail. But in microservices a failure of single service will cause to disable some functionalities, but it will never break the entire system.

Scaling	In monolithic system, if we want to scale we have to scale the entire system. But in microservice system, developers can scale each of the services as required.
Organizational Alignment	Unlike in monolithic systems, microservices organizations can have smaller teams. Each team can work on single microservice; they will have to decide the technology stack that they should use. The internal decisions that they made will not affect the entire system.
Optimizing for Replaceability	In monolithic systems it's hard to remove or upgrade existing components which are obsolete. For example, if developer wants to update system to latest java version, then he will have to stop the system, make necessary changes and test all the available services. But in microservices we can remove and replace services without much fuzz, and we can upgrade individual services, it won't make any affect to the rest of the system.

Table 1 Key benefits of microservices over monolithic systems

2.2 Service oriented architecture vs Microservice architecture

Most of the time both of the service oriented architecture and microservice architecture seems similar to each other, but there is a distinct difference between them. Service oriented architecture is a set of services which collaborate with each other to provide some set of capabilities. The main idea behind the emergence of SOA is to combat challenges of large monolithic systems. But unlike in microservices, services will still run in the same environment. SOA lacks a way to define a clear boundary and a size for services. But because microservices are living in its own space, it's easy to define a clear boundary and a size for the services. In one sense, microservice architecture can consider as a subset of SOA.

2.3 Security in Microservices

Security of any kind of computer system consist of policies and best practices to prevent misuse of the system and unauthorized access [14]. The necessity of a proper protocol increase by the fact increasing number of attacks and security breaches. We need to think about the protection of the data in rest and while data is transferring between different systems. There are various well established practices for system security in monolithic systems. In microservice architecture we have to think about a new set of secure and efficient best practices because of the radical change in the system architecture. For example, in monolithic systems developers were able to trust messages coming from other internal services, but in microservice architecture messages coming from other internal services can't trust any more. Table 2 lists security issues that we will have to face in microservice architecture.

Security Issue	Description
Many small services	In monolithic systems we have a single large system to secure. Now we have different systems. Now the attack surface has increased by a large factor.
Communication between services	In monolithic architecture services are communicate with each other locally using method calls and OS processes. But in microservice architecture services call each other via API calls using technologies like REST. So there is possibility of system breach and attacker will be able to call services behalf of internal services. And architecture should be rich enough to protect the system from this kind of vulnerabilities.
Built with different technologies	A limited number of languages are using in monolithic systems. But in microservices there will be services with different technology stacks. Even if they are in same technology different services will run on different versions of the same technology. So developers should find vulnerabilities in each of them and they have to patch them. If number of services is large, then this process is really difficult.
Autonomous teams are developing microservices	In monolithic systems there is a separate security experts and they made necessary risk management, security controls and security testing. But now teams are autonomous, there can be security experts in the team, but there is no any central security team who are responsible for the entire system security.

Table 2 Security issues in microservices

Next when searching a solution for above issues there are couple factors that we should focus on. Most of these things are not specific to microservices, these are common to any kind of security architecture. When ensuring security of a system we should focus on two measure, prevention and detection. Prevention is to block security breaches and to block damaging to the system. The next measure is detection, because we can't prevent every kind of attacks. Developers will implement logs and intrusion detection systems to detect abnormal system behaviors.

While following these measure we should try to balance confidentiality, integrity and availability of data in the system. Confidentiality means data should be visible to only to the user with rights. Integrity means consistency and trustworthiness of the system and data. Availability means a functioning system without any software conflicts and supporting required bandwidth [15]. Achieving all these three properties at once is impossible. But the architecture should capable to balance all these three properties.

Further we can divide the security perspectives based on system security and application security. In system security perspective we will focus on things like all the services in the system, and setting up firewalls, intrusion detection systems. The application level security can be divided into end user interaction and service to service interaction. In the end to end user interaction, security will be measured from one end point of the user to the other. In service to service interaction, focus will be to improve security between service to service communication.

2.4 Authentication and Authorization in Microservices

Authentication is the process of confirming a user who he says is. Authorization is the process of deciding what user can do and don't. In monolithic systems, this process handles mostly by application by itself. In a security perspective these two operations are the most important processes in any kind of system. There are various ways that we can handle user authentication as lists in table 3,

Authentication/Authorization mechanism	Description
Using a local store	Username, passwords store in the local machine database or file. Then the local machine itself access these resources and verify the user.
Distributed Lightweight Directory Access Protocol	LDAP server will be used; it can be a server hosted externally or server which shared among multiple local systems.
Delegating access	Technologies such as SAML, OAuth, Kerberos can use to delegate user access. Most commonly these kind of technologies use in single sign on implementations.

Table 3 Authentication and authorization mechanisms

Among the above three methods, access delegation is a very common technique used in these days, because of its simplicity and secure nature. In this method, when a user tries to access a resource he will be redirected to an identity provider. Identity provider may ask username and password from the user for authentication. After the user confirmation, the identity provider will redirect user to services he requested. Identity provider can be an external service such as OpenAM or Facebook. Identity provider may use Lightweight Directory Access Protocol(LDAP) or any other directory system to store user data.

2.5 OAuth 2.0

OAuth is an authorization framework, that can use to pass user information and other kind of resources from one location to another without providing user's username and password. So the user details are stored in an identity service and local servers will not have to store them. Local server will access this information via identity server after user approving the access delegation process. In this process identity server returns an access token which can issued to a third party.

OAuth 2.0 is especially designed to work with HTTP protocol. In OAuth, four roles are involved with this authentication process, those are lists in table 4, [16]

Role	Purpose
Resource Owner (RO)	The user
Client	The web or mobile app
Authorization Service (AS)	OAuth 2.0 server
Resource Server (RS)	where the actual service is stored

Table 4 Roles in OAuth authentication process

The flow of granting user access in OAuth process is as following, (figure 7 shows this flow in a diagram)

1. The application requests authorization to access service resources from the user
2. User authorizes the request
3. Client request an access token from Authorization server by passing his own identity and authorization grant
4. If the application identity validated and grant is validated, server will return access token to the client
5. Application request resource from the resource server by presenting the access token
6. Resource server asks authorization server, if access token is valid, it will return necessary information. And finally this will give access to the protected resource

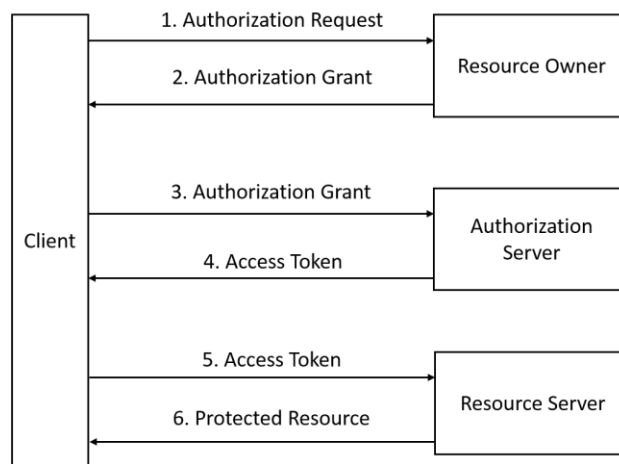


Figure 7 OAuth user authentication

OAuth authentication system has couple of security threats as lists in table 5: [5]

Security Threat	Description
Token manufacture/modification	An attacker may generate a bogus token or modify the token contents (such as the authentication or attribute statements) of an existing token, causing the resource server to grant inappropriate access to the client.
Token disclosure	Tokens may contain authentication and attribute statements that include sensitive information.
Token redirect	An attacker uses a token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be for it.
Token replay	An attacker attempts to use a token that has already been used with that resource server in the past.

Table 5 Security threats in OAuth protocol

Threat Mitigation with OAuth protocol [5]:

- A large range of threats can be mitigated by protecting the contents of the token by using a digital signature or a Message Authentication Code (MAC).
- To deal with token redirect, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token.
- The authorization server must implement TLS. To protect against token disclosure, confidentiality protection must be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.
- Cookies are typically transmitted in clear. Thus, any information contained in them is at risk of disclosure. Therefore, bearer tokens must not be stored in cookies that can be sent in the clear.

2.6 OpenID

Not just authentication if the client application needs to take extra information about the user OAuth protocol will not be enough. In this case OpenID protocol can be used. In this protocol the OpenID Connect Provider (OP) returns an ID Token along with the Access Token to the client. Both of these two protocols seem similar, OpenID is based on OAuth protocol. But there are some differences such as, for example, OpenID defines a userinfo endpoint, but OAuth doesn't have such a thing. Steps of OpenID authentication flow are lists below,

1. Client request access to the resource server by calling Authorization Server
2. Authorization server redirect to allow user authentication

3. After validating authorization server will return an Access Token and ID token
4. Now the client can use the ID token to enhance the user interface
5. Then client can send access token to resource server
6. Now resource server can respond with the data client needs

2.7 JSON Web Tokens (JWT)

JSON web token (JWT) is a URL safe self-contained compact container to transport data between two or multiple parties. There are many applications which are using JWT. OpenID is one such application. JWT can be signed by using public/private key, so the consumer can verify the source later. JWT's are known as compact because its small in size and it can be included in a URL or in a REST request header. JWTs are known as self-contained because it contains all the information required to understand details in JWT.

JWT's can be used for user authentication and information exchange. There are two type of JWTs, JWE (JSON web encryption) and JWS (JSON web signature). In JWE content of the JWE is encrypted, but in JWS the content is signed using a private key. So external entity can verify this using the public key [17]. A JWT contains three sections, each section is Base64 encoded values separated by dots. These values are

- Header
- Payload
- Signature

So a typical web token will look like this,

```
xxxx.yyyyyy.zzzzz
```

In here the first part represented by xxxx is header information, it contains the hashing algorithm and the token type.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Then the next part is payload, which may contain claims (Data about the content) and additional metadata. There are several reserved claims and there can be custom claims. A claim will look like,

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Table 6 lists some of reserved claims in JWTs which are more important. [18]

JWT Claim	Use of the JWT claim
sub	This parameter represents the subject who owns the jwt
aud	This parameter represents the intended audience of the token
exp	This parameter represents expiration time on or after jwt should accept
iat	This attribute represents the time the jwt issued
jti	This parameter represents a unique identifier for the JWT. This can be used to prevent replay attacks

Table 6 JWT claims

Then the final part of the token is the signature. This can be used to verify that the token hasn't changed by a third party. To generate signature, encoded header and payload is signed using the algorithm specified in the header. All these three base64 encoded values are concatenated using dots in the token. Then this token will include in the each and every subsequent request headers as shown in below.

Authorization: Bearer <token>

Because this token include all required information, it will not store in memory of the server. Hence this is stateless mechanism. There are other kind of user authentication mechanism using SAML and SOAP. But use of JWT is much easier than using SAML based authentication. SAML based tokens have to parse, create object structure and user authentication according to SAML is poor for performance and this whole process is very complex.

2.8 Token based authentication and session handling

Nowadays most of the APIs and web service are using token based authentication. This mechanism use tokens similar to JWT for the authentication process. In past it was more common to use cookies for user authentication and session handling. In this case, sessions are stored in the server since HTTP is a stateless protocol. Following figure 8 is the simple procedure of user authentication using cookies. There are several problems with cookies based authentication, those are lists in table 7, [19].

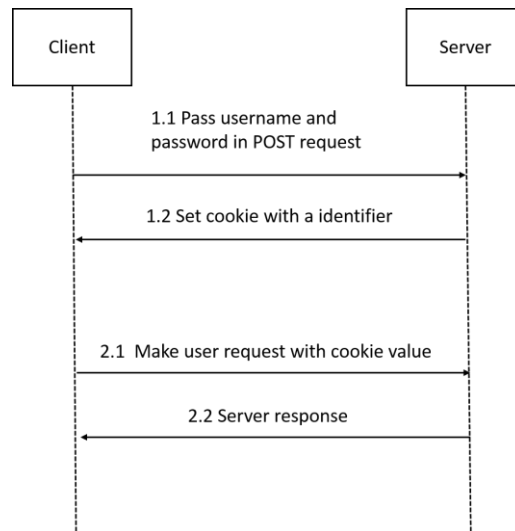


Figure 8 Cookies based user authentication

Problem	Description
Storage	When a user login to the server, server will have to create a temporary session somewhere in the memory. If there are too many users accessing to the system at the same time, then there will be higher overhead to the system.
Scalability	Because the sessions data is stored in memory, in case of scaling the system this data has to be replicate
Cross origin Resource Sharing (CORS)	When the application attempts to grab data from other sources, we can come up with forbidden request problems
Cross site script forgery	Users can already logged in with some other site and that information may steal session information saved in browser

Table 7 Problems with cookie based authentication

Token based authentication is stateless, so it gets rid of many problems mentioned above. JWT tokens can be used for this authentication process. Following figure 9 show the process of the authentication using JWTs.

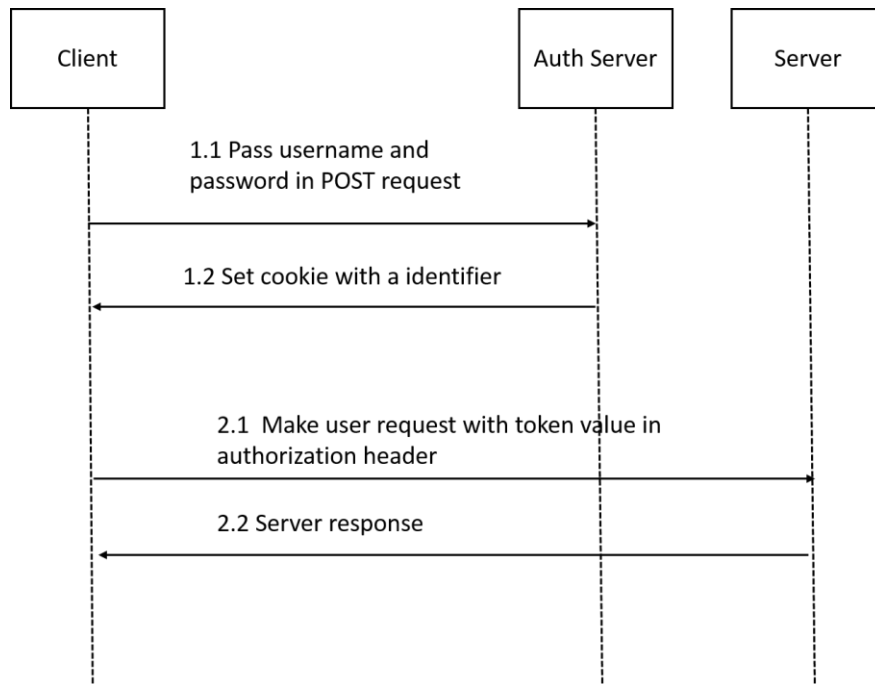


Figure 9 Token based user authentication

Advantages of token based authentication is lists in table 8.

Advantage	Description
Tokens are stateless and scalable	Tokens contains enough information for authentication. Because this is stateless, application can scale as required.
Ease of generation	Tokens can generate from anywhere and then it can verify from somewhere else. This allows to decouple these two processes cleverly.
Fine grained control	All the necessary information about permissions and user roles can be included in the token.
Standards Based	Based on well-established standards

Table 8 Advantages of token based authentication

2.9 Kerberos

Kerberos is a computer network authentication protocol; it allows for users to communicate with each other over a non-secure network in a secure way. Kerberos perform the user authentication process by exchanging tickets between each of the parties involved. Kerberos is using a symmetric key cryptography and rarely using public key cryptography. By combining all these things Kerberos prevents eavesdropping and reply attacks in communication. In Kerberos protocol, Key Distribution Center (KDC), which brings client and server together. One of the important property

is, that there is no communication between KDC. Here each of these parties has their own keys, and the key also exist in KDC. First client passes his key as an authenticator after encrypting to the KDC. KDC can verify the key, because he already has the user's key. After the verification KDC will pass Ticket Granting Ticket (TGT) to the client.

When user want to communicate with file server, he passes the TGT and the message he need to communicate with the server to KDC. KDC verify the user information and then he passes a new ticket to the client after signing a ticket by the key of the server machine. And client pass this ticket to the server and then server will attempt to verify the ticket. Server will grant the communication on successful verification of the ticket. Advantages and disadvantages of Kerberos protocol are lists in table 9,

Advantages	Disadvantages
Users doesn't have to pass his password via network	Complexity of the protocol
Credentials has a limited life time	Kerberos is originally developed for LAN networks, not for public internet
Centralized KDC server makes it easy to control the entire process	Kerberos needs synchronized clocks, it's rather easy to do this synchronization on a LAN network, but doing it in a public internet is really hard.
Once a client and server authenticated with each other they don't have to renew the ticket until token expire	Kerberos is to provides mutual authentication, both the user and the server verify each other's identity. So this means we will have to store necessary keys in client machines. It's really difficult to distribute these keys over public internet.

Table 9 Advantages and disadvantages of Kerberos protocol

2.10 Securing service to service communication

In microservices service to service communication is another important aspect that we have to focus on. In monolithic systems this can be done as local method calls, but in microservices we have to handle this as inter-process communication. This communication can be either of synchronous or asynchronous communication. In synchronous communication we can use REST based protocols. In asynchronous communication we can use technologies such as RabbitMQ. A microservice system may use a combination of these methods to communicate with each other.

In service to service communication the user information should pass to backend services. Here we can make couple of assumptions. One assumption is we trust the backend services, so we can just merely pass user information among backend services. But if an attacker access to the system then he can act as a man in the middle and change the data. There are different security mechanisms that we can implement in order to support secure service to service communication, such as HTTPS Basic authentication, Using SAML or OpenID Connect, Client Certificates, Kerberos, HMAC over

HTTP and API Keys. Some of these methods will consider in design phase to secure the service to service communication.

2.11 OWSAP security standards for system vulnerabilities

The open web application security project (OWSAP) is an organisation which provides information about application security. This organisation has published a set of top 10 most critical vulnerabilities that can be found in web applications. These vulnerabilities occur very frequently in web applications and those can be exploited easily. OWSAP introduces following top ten security vulnerabilities as lists in the table 10 [22],

Vulnerability	Description
A1- Injection	Injection flaws, such as SQL, OS and LDAP injection occur when untrusted data is sent to an interpreter as part of an application request. So, the backend application will execute unintended processes which did not expected by developers.
A2-Broken Authentication and Session Management	This happens if functions related to authentication and session management are not implemented correctly, allowing attackers to compromise the passwords and keys.
A3 - Cross Site Scripting (XSS)	This occurs when an attacker can execute malicious scripts to the server and then later he can hijack user's sessions and redirect users to malicious of phishing sites.
A4 - Insecure Direct Object References	A direct object reference can occur when a developer exposes a reference to internal implementation object, such as a file or database key. So, then the attacker can later use this reference to access the system directly.
A5 - Security Misconfiguration	System servers, directory servers and databases should setup with correct security configurations defined for the specific component. Additionally, software should keep up to date.
A6 - Sensitive Data Exposure	Many web sites are not protected sensitive data like passwords and credit card numbers. So, an attacker may retrieve them from system databases or any other system resource.
A7 - Missing Function Level Access Control	Some applications verify functional level access rights in UI level, but if the same verification is not done in the backend then an attacker can easily exploit the backend and access functionalities without proper authorization.

A8 - Cross Site Request Forgery (CSRF)	CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
A9 - Using Components with Known Vulnerabilities	If vulnerabilities exist in components such as libraries, frameworks and other software modules, then attacker will be able to exploit the backend component and access to the backend system.
A10 -Unvalidated Redirects and Forwards	When redirecting users to other websites untrusted data may use to determine the destination pages. Without correct validation, attacker can redirect the users to phishing or malicious sites.

Table 10 OWSAP top 10 vulnerabilities

Chapter 3: Design of Solution

Monolithic systems are hard to maintain, develop and scale. As a solution for these inherent problems of monolithic architecture, microservices architecture has emerged. In microservice architecture there are different service running independently on separate containers. Microservice architecture allows different parts of the application to be develop, deploy and scale independently. However, now we have a new set of security problems to handle, like securing service to service communication.

This research focus on handling authentication, session handling and service to service communication in microservice architecture. Here I will design two backend architectures for service authentication and session handling. Then a comparison between these two architectures will be made in Results and Evaluation chapter. Then a command line based and GUI based interfaces for this system will be implemented. The details about the implementation will be presented in chapter 4.

3.1 Authentication Server

Authentication is a one of the most important operation in any kind of computer system. Authentication is the process of confirming the user who he says he is. User authentication can handle using either or internal or external authentication server. One possible option is to use Local LDAP server and other options is to use access delegation services like OpenID, OAuth, SAML or Kerberos. Use of Local storage such as LDAP is suitable for computer systems running in local environment. But if any system using such a mechanism for public internet, then there we have another set of problems to solve such as securing authentication server, creating backup servers to store user information and implementing account creation process. But we can get rid of all these burden by using external access delegation systems.

As already mentioned there are several options, when choosing an access delegation technology. Among these technologies Kerberos is one of the popular mechanism in academic studies. But using Kerberos in public internet is not a practical approach as already mentioned in chapter 2. Using SAML based authentication is another option, but most of the time it uses in enterprise single sign on mechanisms, such as within enterprise, enterprise to partner and enterprise to cloud services. So this mechanism will not be viable for the microservice system.

OAuth and OpenID is designed for internet scale access delegation and authentication. Most importantly this mechanism is simpler than all other access delegation methods mentioned above. By using OAuth, we can get user information which stored in another system. Here the system will get credentials and then those credentials will pass to the OAuth authentication service, then it will return an access token. Then the token can use by the backend microservices to verify the users.

In OAuth there are several types of tokens. One of them is by reference tokens or standard access tokens and other one is by value tokens or JSON Web Tokens (JWT). By reference access tokens doesn't contain any information it's just a string, returned by the authentication server to the client. On other hand by value tokens contain information. One of the drawback of token based authentication is the exposure of token information to the public because they are not encrypted. To mitigate this drawback, we can use the access tokens outside the microservice system and internally we can use JWTs. After authenticating with the OAuth service user will pass the access token to the microservice system, then microservice system will validate the access token and it

will retrieve a JWT token and then it will pass among microservices. In the first backend microservices will handle the token exchange process and in the second backend reverse proxy will handle the token exchange process. There are several advantages of this system. This system will not expose user credentials to the microservice system, this system will not expose unencrypted JWT tokens to the outside and another advantage is this whole system is based on well-defined standards.

3.2 Architecture

Following sections contain two backend architecture designs for user authentication. The first architecture will authenticate users by communicating with an authentication server directly. In the second architecture, a reverse proxy server will be used for authentication. Both of these systems will use OAuth as the authentication server. In the following sections, only the system design will be presented, specific technologies that should be used to implement it are mentioned in chapter 4.

3.2.1 Back-end 1 – Authenticate users by microservices

In the first backend design, user authentication will be performed at the microservice level. Here, the client will try to access backend services by using an OAuth authentication server. First, the client will authenticate with the authentication service and pass the authentication token received from the authentication server to the backend. In the backend, each microservice has separate user authentication modules. Figure 10 shows this back-end design. This design also contains several microservices, a database, and a caching database, each of them is running in separate containers. The purpose of using a caching database will be explained later.

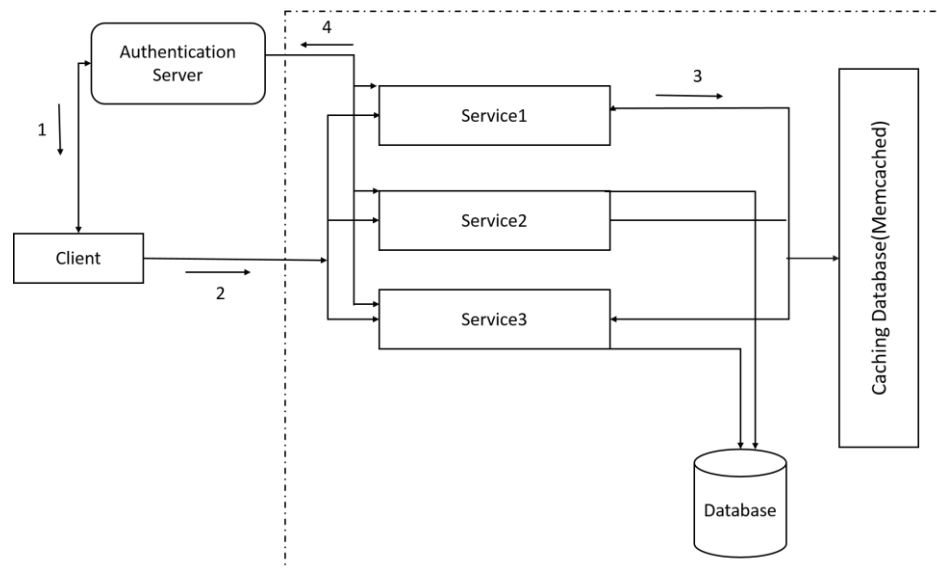


Figure 10 Back-end 1 Authenticating users in microservices

1. First user will pass his credentials to the authentication server. And after validating the user, authentication server will pass an access token to the user.
2. Then user will pass the access token to the backend service.

3. Then microservice will check whether access token exist in caching database. (This is explained below)
4. If access token doesn't exist in caching database, then microservice will pass the access token to the OAuth server to validate. On succesful validation microservice will return correct response to the client.

Figure 11 shows the sequence diagram corresponding to this flow.

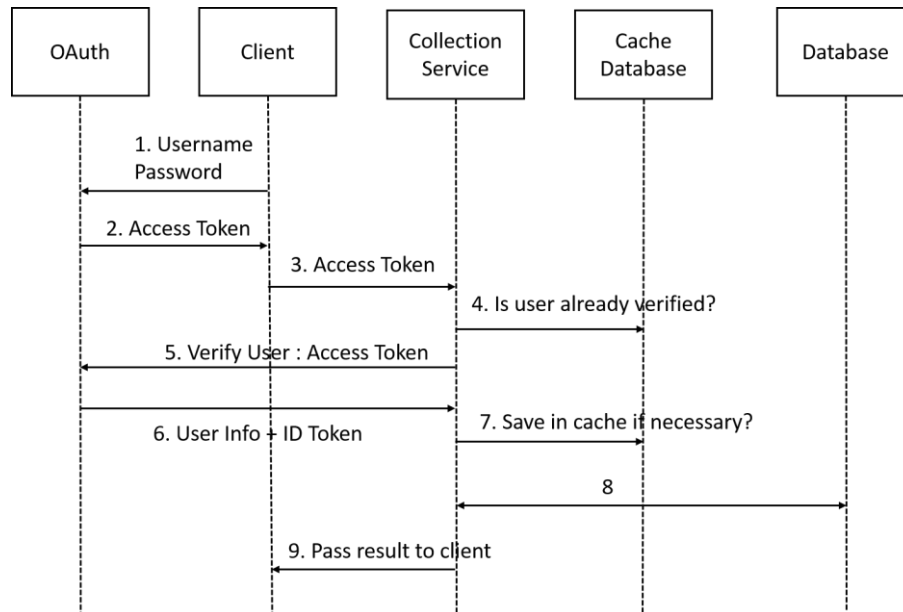


Figure 11 Back-end 1 Authenticating users in microservices - Sequence diagram

3.2.2 Back-end 2 – Authenticate users by a reverse proxy

In this architecture, user authentication will be done by a reverse proxy. The reverse proxy will authenticate each user's request and if its valid, then it will pass the request to other microservices within the network. One of the major difference in this system and Back-end 1 is moving the authentication logic from microservices to reverse proxy. This improves system maintainability and adds several advtages to the system. Figure 12 shows the architecture of this system.

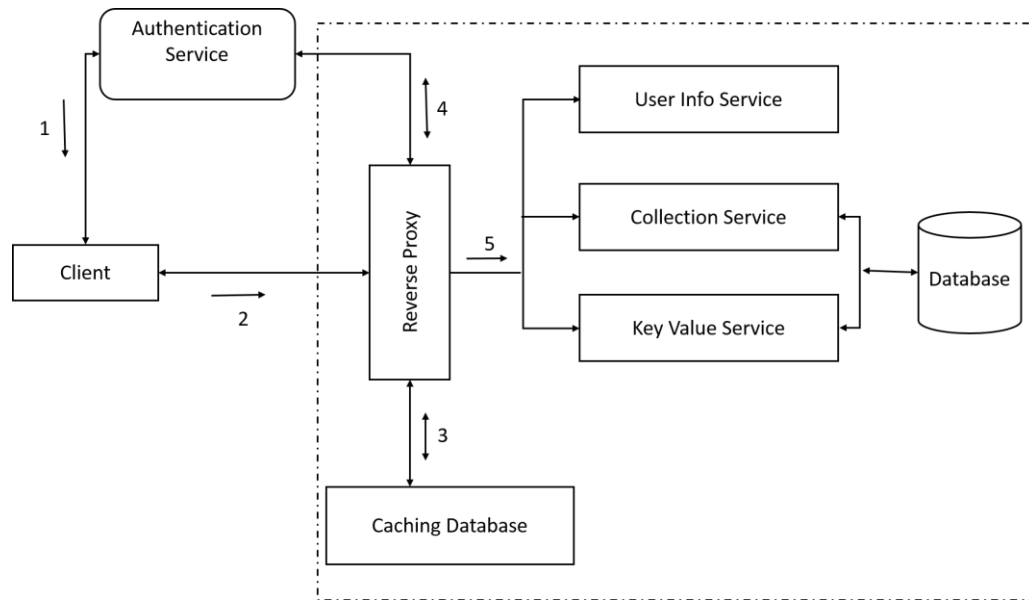


Figure 12 Back-end 2 Authenticating users in microservices

User Authentication process;

1. User connect to the authentication server and pass user credentials and after validating the user, it will return an authentication token back to the user
2. User will pass the access token to the microservice system.
3. Reverse proxy check user information in caching database. (This will be explained later.)
4. Reverse proxy pass the access token to the authentication server and validate.
5. On successful validation, reverse proxy will pass this information to internal microservice and response will be returned to the user.

Figure 13 shows the sequence diagram of the user authentication flow,

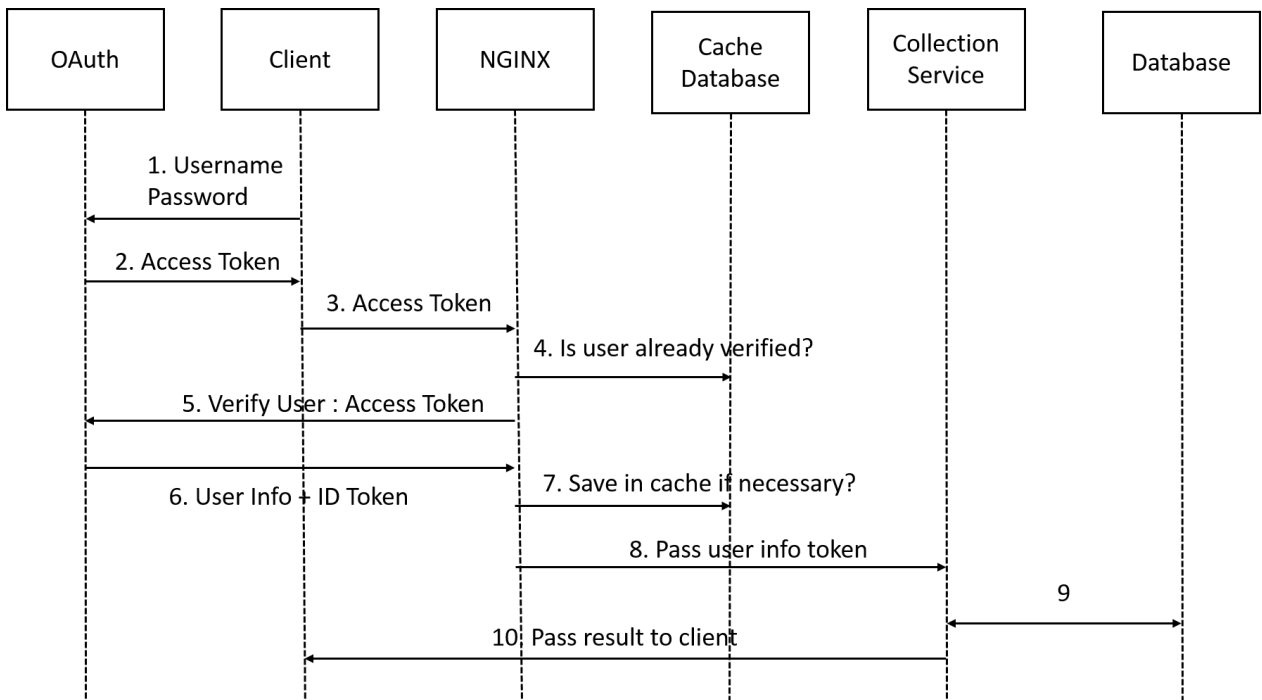


Figure 13 Back-end 2 Authenticating users in microservices - Sequence diagram

3.4 Caching database

A caching database is used in both of above architectures to improve system performance. The access token which passed by the is a one-time token. So if user accessing another service then he will have to regenerate the token. The idea of using the caching database is to prevent this. And even if user could generate unlimited number of new access tokens without a performance issue, reverse proxy/microservice will have to keep validating the same user with by communicating with external OAuth server. So idea is, once a user login to the system reverse proxy/microservices will validate the token using OAuth servers. On successful validation the id token and user info token will save in cache database. To implement the cache database, we can use an in-memory database like Memcached or Redis.

In the first back-end microservices perform user authentication by itself. When a user login to the system first the microservice will check the availability of the user information in the caching database, if it's not available then microservice will authenticate the user. And after that user information will save in the database. So when user accessing another microservice, user information is already saved in the caching database and the microservice doesn't have to authenticate the user again.

In the second architecture reverse proxy performed the user authentication. In this case microservices are not directly connected to the caching database and reverse proxy is connected to the caching database. When a user logged in, reverse proxy will first check user information in the caching database, if it doesn't exist then it will pass access token to the OpenID/OAuth server and if its valid then user information will be saved in the caching database and user information will pass to microservices. On subsequent requests, reverse proxy can get the user information from the caching database and pass it to internal microservices. There are three types of token we received from the authentication server. A new access token will return after passing the access token

returned from the user. Then a user information token and an id token. The caching database will store all these three tokens and later these tokens should delete or invalidate on user logout.

3.5 Reverse proxy to service communication

One assumption that we can make is all the service calls made from inside the system can be trusted. But depending on the sensitivity of data we will need secure service calls. Otherwise if someone penetrates the system, then he can act as a man in the middle. He will be able to read and change the data. In this system, only internal communication happens, between reverse proxy to microservice and microservice to databases. Among these things, reverse proxy to microservice communication exchange sensitive data between each other. If an attacker could penetrate the system then he can send invalid user tokens to backend microservices, so as an example if there is a service to withdraw money from a bank account, then an attacker will call this service directly with a bogus user information token. Because of this reason, securing this communication is important. There are various ways that we can secure this service to service as listed in table 11 [6]

Service to service communication secure method	Description
Use SAML or OpenID Connect	Each service will communicate with each other using any of these services. This still imposes a problem when storing username and password. Client will have to find a secure way to store these details.
Client Certificates	Here each client has a X.509 certificate installed, which is used for communication between server and client. This will introduce a problem into the system when managing certificates.
HMAC Over HTTP	In HMAC content of the request is hashed with a key and this value is sent with request. Primary problem with this approach is sharing the private key among services. Most probably it will have to be hardcoded in each service. Then it will be difficult to revoke the private keys.

Table 11 Securing service to service communication

All of the above methods has problems as already listed. One other simple solution is to sign the user token by the reverse proxy before passing to internal microservices. Then internal microservices can verify the signature and if signature is correct accept it or else they can reject it. Here the reverse proxy will have a public/private key pair and public key of the reverse proxy will store locally in the microservices and when passing a token reverse proxy will first sign the token and it will pass to internal microservices. They will verify the token using locally stored public key.

This is a simple solution but storing public key locally will introduce several problems such as difficulty in revoking the certification. Furthermore, if we later decided to use multiple reverse proxies to balance load, then certificates of all these servers will have to store in microservices. A solution for this problem is to use an external public key store. So the key store will hold all the public keys, users can request public key from the key store and use it to verify the token. Then system again expose to the same problem we tried to solve, man in the middle attacks. Now attacker can act as the public key store and initiate attacks. A solution for this is to generate a public private key pair in the key store, and it will sign all the public keys with its private key before passing to other services. And all services will hold the public key of the public key store locally and they can verify the token and the public key. The advantage of this system over previous solution is, its extensibility. Figure 14 shows this process in diagram.

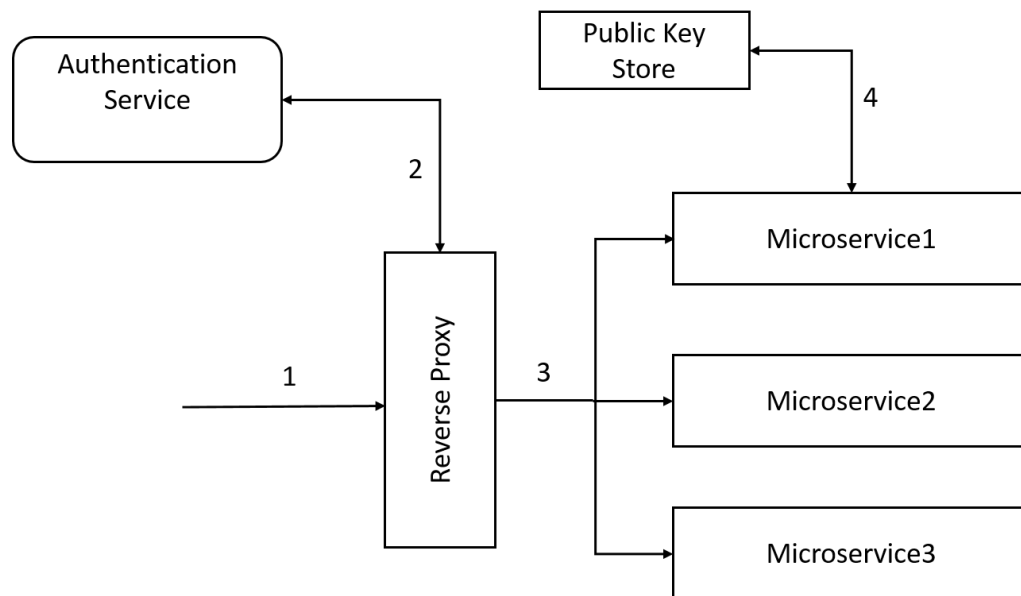


Figure 14 Reverse proxy to microservice communication

1. User send a request to access a microservice
2. Reverse proxy validate user token by contacting authentication server
3. Reverse proxy sign the token with its own private key and pass it to microservices
4. Microservice ask the public key of the reverse proxy from key store. public key store will pass the signed public key of reverse proxy. Then the microservice will validate these signatures.

3.6 Handling user sessions

Chapter 2 mentioned several drawbacks of using cookies based authentication system. It has problems with storage, scalability, cross origin resource sharing and cross site script forgery. Token based user sessions are one alternative to mitigate these problems. Token based authentication is prominent everywhere on the web nowadays. Most of the major web APIs like Google and Facebook are using token based authentication.

In this system a user session will create once the user logged into the system. On user login, information about his session will fetched from authentication server. Authentication sever will

return this information in an id token. The reverse proxy or microservice will store the token in cache. On subsequent requests, user will pass access token to the microservice system. Then the reverse proxy or microservice will get the corresponding id_token from caching database and it will pass among other services. The microservices can get information about user session from this token. Here the id_token act as a saved user session, on the termination of the user sessions we will have to remove this id_token from the database. One drawback of this mechanism is, if a microservice wants to keep user specific data temporary then the microservice will have to use a local database.

3.7 Terminating user sessions (User logout)

Another important process is user logout service. When a user logout the system, we have to terminate user session. Termination of user session can be done in several ways.

1. User terminate user session by calling logout service
2. System terminating user sessions on expiration of id token

In the first situation client will pass a logout service request to the microservice system. Then the reverse proxy has two tasks to perform. First the reverse proxy should delete data related to the user in caching database, then it has to inform internal services about the user's logout action. Caching database contain information about the logged in user's access token and other related tokens. So first we should delete this information from the caching database. Otherwise user will be able to access system even after logout. But instead of deleting user information, it would be much better if we just update the caching database tokens as invalid. The reason is, if user made a request again with the same access token, then we will have to check it with authentication service and this is an unnecessary service call. So if we just update the access token as invalid in caching database, it would be a good performance tweak.

Informing microservices about the user logging out is important because of several reasons. The microservices can hold unsaved data related to the user, so after logging out, that data should delete or remove from memory. If there are any open connections to external services created by the microservices, those connections should terminate. If there are any streaming services initiated by the backend microservices, then it should terminate. We can handle user logout in several ways. First one is that the reverse proxy can store locally about the services that user is logged in. So the reverse proxy can iterate among each of these entries and inform to services about the user's action. But the problem with this approach is, some service will not interest in this event. And if service A call to service B then this event should pass to service B also and reverse proxy is unaware of that service A to service B communication.

Another solution to this problem is to use a publish subscriber mechanism. Microservices who interested about the user's logging out will be subscribe to the publish subscriber queue. And on the event of user logging out reverse proxy will inform to the publish subscriber system about the event. We can implement pub sub system locally in the reverse proxy server, but this will increase the overhead on reverse proxy server. So it's better to use an external publisher subscriber system such as ZeroMQ or RabbitMQ. All the microservices who interested about user logging out will subscribe to message queuing system, and on the event of user logging out reverse proxy will pass this event to the message queuing system. One important point is, that the user logging out is not a

synchronous process anymore, its asynchronous. There are several advantages of this system as following,

1. There is no any extra overhead to handle user logout for reverse proxy
2. User logout event will get only by the services who interest about this event
3. In overall system will not have to wait till the completion of this service because of its asynchronous nature.

Figure 15 shows the system architecture to handle user logout,

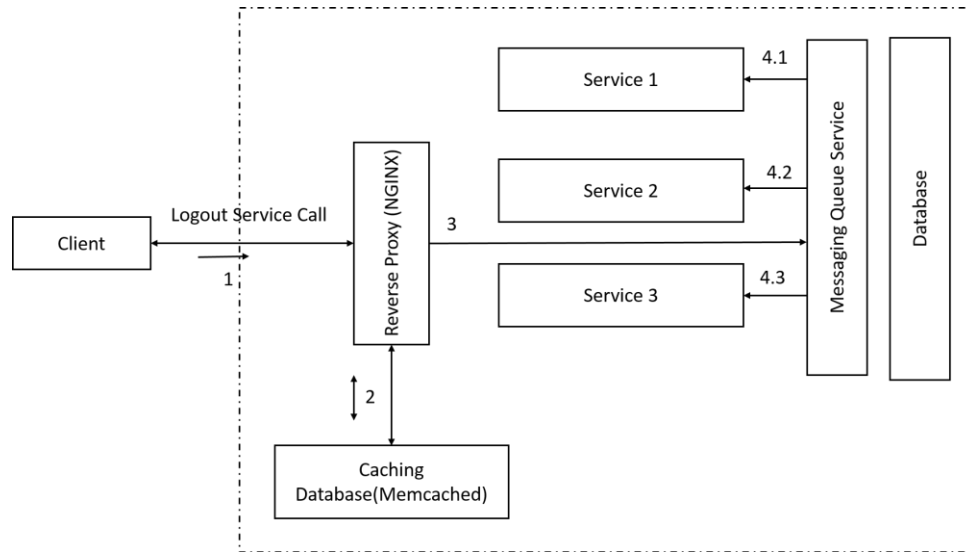


Figure 15 Architecture for user logout

1. User request to logout
2. Reverse proxy invalidate the caching database entries associated with user
3. Reverse proxy pass the event of user logout to the messaging queue service
4. Pass the event to services subscribed to the queue service

Previously system design will handle user logout initiated by user. But in some occasions, system has to terminate user sessions. There can be various reason to terminate session such as if a user is idle for a longer time, if a user is doing a suspicious action or on the situation of user token expiration. In this design handling the user logout due to the token expiration will consider. Other situations will consider as future work. There is a claim named exp in the id token that we received from the authentication server. This claim uses to identify the expiration time on or after jwt should accepted. So in this case user session will have to terminate automatically after exceeding this time. This can automate easily by using scheduled tasks. After the scheduled time an event will pass to messaging queue service informing user logout event.

3.8 Securing the system

3.8.1 Secure Communication with HTTPS

In this systems, there are couple of endpoints which expose system to outside environment and it creates some security risks for the system. On user login, user transfer his access token to the microservice system, external party can intercept this communication and get the access token and pass this token with a different service call. Other than that, when microservice system authenticating the user token via third party, attacker can intercept communication and get the id token and user token values. Finally, if attacker could penetrate the system he will be able to monitor the requests passing among the services within the system.

For the first two situations we can use HTTPS connections to secure the communication. So client will have to use HTTPS connection to pass access token and the backend servers will use HTTPS connection to validate access token with OAuth/OpenID servers. This process will secure the information exchange and it will prevent the in the middle attacks. Finally, we can implement HTTPS to secure service to service communication. One service call from the user will cause to call several other microservices and database calls. But if we try to use HTTPS for all these requests then the performance of the system will degrade significantly.

3.8.2 Token Validation

Tokens which returned from the authentication servers should validate to use them in a secure way. JSON web tokens contains a signature to prevent malicious attempts of changing the token content. OAuth service will sign JWT token using symmetric algorithm or asymmetric algorithms. In this system we are using asymmetric algorithm RS256 to sign tokens. Authentication server sign the token using its private key then the microservice system will validate it using public key of the authentication service. Then the validation process will be done in the reverse proxy server. There are various ways to get the authentication server public key, such as JSON web key and using PKIs, but in this system we will use a static configuration to get the public key. Sometimes, some service will return tokens without a signature and the signature algorithm will set to none. Based on the sensitivity of the data, system should either accept or reject such kind of tokens.

Furthermore, a jwt token contains several other claims that can use to validate. Some of these claims has already described in the second chapter. “sub” – claim can be used to check who the subject own, “aud” claim can be used to check the intendent audience and finally “exp” and “iat” tokens can be used to identify the time token issued and the time that the token will expire. All these claims can be used by the reverse proxy to validate the user tokens. Another important claim in the JWT tokens is nounce, this value can use to prevent Reply attacks. Reverse proxy and microservices will store nounce values of already received tokens and if the microservice system received another token with same nounce then it will reject.

3.9 Summary of chapter

This chapter presents a detailed description about the proposed solution to the research problems with diagrams. A separate authentication server based on OAuth and OpenID will use to handle user authentication. On user login, authentication details will pass to the backend. Two back-ends

developed to validate and distribute authentication details among other microservices. One of them authenticating users at microservice level and the other authenticate users by a reverse proxy. Tokens are used to manage user sessions and a separate architecture developed to handle user logout. Several steps are mentioned here to improve system security, such as using different token types, signing tokens before passing to other services and validating tokens using various claims to prevent attacks to the system. Additionally, a caching mechanism and asynchronous service calls used to improve system performance.

Chapter 4: Implementation

In previous chapter, two back-end architectures designed to handle user authentication and identity distribution. In first design, microservices authenticated user requests using OpenID/OAuth server. And in the second design a reverse proxy handled the user authentication process. Both of these designs implemented and evaluated. A client application is also developed to demonstrate the functionality of this system. Both of the microservice architectures implemented in Docker environment.

4.1 Backend services

The system implemented using couple of example microservices. The purpose of implementing the services is to demonstrate the functionality of the system. In the backend four microservices are implemented using PHP and NodeJS. Table 12 lists the services implemented in the back-end. All of the services, should invoke by the front end application using REST based HTTP requests. And all of the microservices except the streaming service will return JSON responses. Most of these services will return {"success": true} parameter.

Service	Purpose
Userinfo service	This microservice implemented using Node.js. The service returns user information fetched from the OAuth server
Collection service	This microservice implemented using PHP. The microservice can use for two operations. To create collections and to list the available collections. The collections are similar to a key value table; they can use later to store key value pairs. E.g. Country_City, Language_Inventor
Key-Value service (Wallet Service)	This service can use to create key-value pairs in the collections created by the collection service. The microservice can use for two functions, to add key value pairs to a collection and to list the available key value pairs in a collection. E.g. Add key value pair Sri Lanka, Colombo to Country_City collection.
Stream Service	This service programmed using Node.js. The service act as a streaming service. Using a web socket the service keep streaming data after user authentication.

Table 12 Services in the backend

To invoke above services user should call the service with a URL with correct parameters. Every service invocation contains service name and some of the service invocations will include an operation name too. The service name is only required for services in second back-end, it is not necessary in first backend, because client connect with it using direct IP addresses. Addition to the

service and operation parameter there can be some extra parameters. Table 13 and 14 lists some of the examples and the use of each service.

Operation	URL parameters
User login	hostname:port/service/login
List collections	hostname:port/service/collection/operation/list
Add a collection named country	hostname:port/service/login/service/collectionoperation/add? name=Country
Add a key value pair to collection country	hostname:port/service/login/operation/add?collection=Country&key=SriLanka&value=Colombo

Table 13 Example operations and rest URLs

Service	Service Name	Operation Name	Extra paramteres
Login	login	-	
Collection	collection	1. Create Collections - add 2. List all the available collections – list	1. name – collection name
Key Value Store	wallet	1. Create key value pairs in collections - add 2. List key value pairs in the collection – list	1. collection – collection name, key – key to insert, value – value to insert 2. collection – collection name
Streaming service	-	-	-
User information service	userinfo	-	-

Logout	logout	-	-
--------	--------	---	---

Table 14 Services in the backend

4.2 Authentication Service

In this implementation we used an external service for OAuth/OpenID authentication. When selecting an OAuth server, there are only few options available. Two of the them are Gluu and OpenAM. Among these solutions OpenAM used for implementation of the system. OpenAM is an all in one access management system. This support for user authentication, federation, single sign on and etc. OpenAM has built based on java, so the OpenAM server runs inside a tomcat server. In this application we are concerned on the OpenAM functionalities related to OAuth and OpenID. All the information related to the user are stored in the OpenAM using a LDAP server. Microservice system access two end points of OpenAM server. Following end point use to authenticate user,

"grant_type=authorization_code&realm=/&code=access_token" – with this http request following http headers should pass to the OpenAM server for user authentication.

Authorization: Basic [base64 value of client account name and password]

Host: [Host Name]

Cache-Control: no-cache

Content-Type: "application/x-www-form-urlencoded"

Following end point can use to retrieve user information,

"/openam/oauth2/userinfo" – with this http request following http headers should pass to the OpenAM server to retrieve user information

Authorization: Bearer [id_token]

Host: [Host Name]

Cache-Control: no-cache

Content-Type: application/x-www-form-urlencoded

4.3 Implementation of first backend

As already mentioned, the system developed as two back-ends. Following is the implementation of the first backend. As shown in the figure 16, there are four microservices in the backend. To invoke microservices, user will have to call each of these services using their IP addresses and port number. Important points to notice about this diagram is microservices authenticating users individually and the microservices are directly communicating with the caching database. Additionally, each of the services and the databases are living in separate Docker containers. Futhermore, in this system authentication logic and the token caching process had to implement in

all the services. Additionally, because there are services which implemented using PHP and Node.js, the same functionality had to program using two languages.

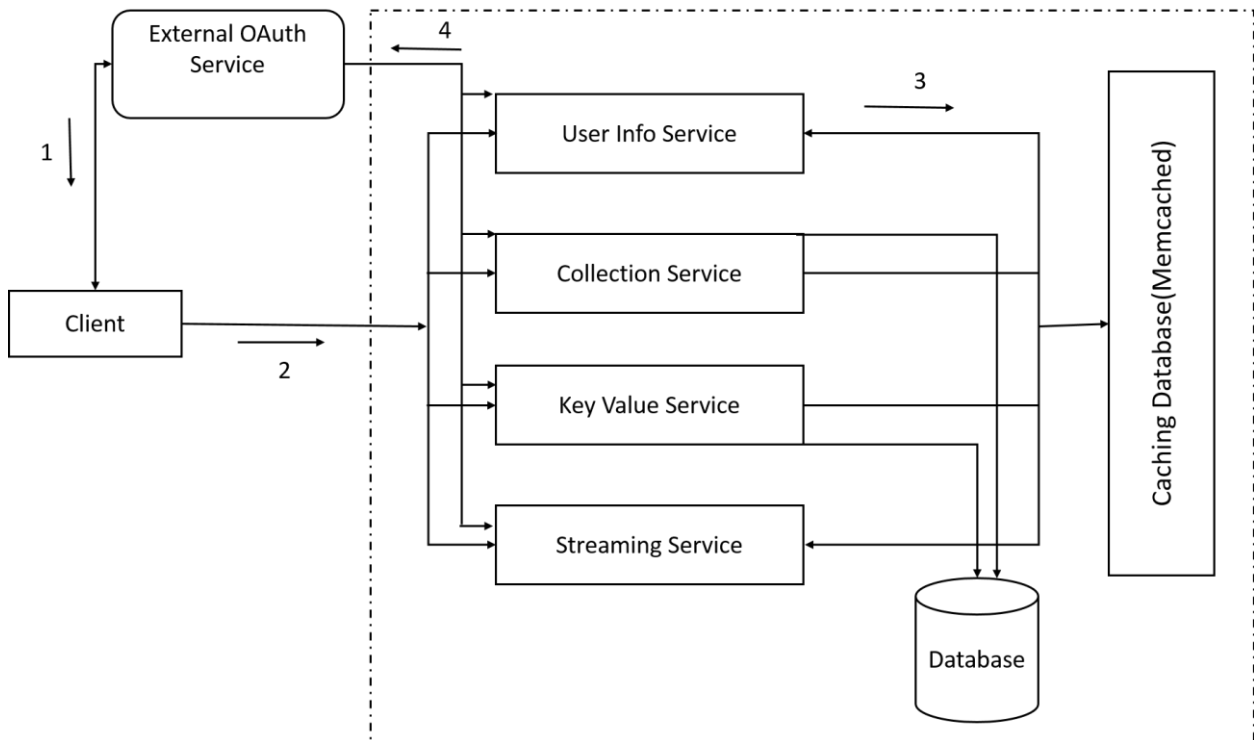


Figure 16 Implementation of Back-End 1

1. Client gets an access token after passing his username and password to the external OAuth server
2. Client pass the access token to any of the services he wants to call
3. The service checks with caching database, is user already validated
4. If user doesn't exist in caching database, then services will validate user with OAuth server
5. After successful validation, user information will be saved the caching database

4.4 Implementation of second backend

Following is the implementation of the second architecture. In this architecture, a reverse proxy is authenticating the users instead of microservices. In this system, services are not directly communicating with the caching database and reverse proxy storing the user information in caching database temporarily. One advantage of this system is, that the users will use the same domain to invoke all the services in the backend. Reverse proxy will decide the service it should invoke based on the requested URL. Figure 17 shows the architecture of this system.

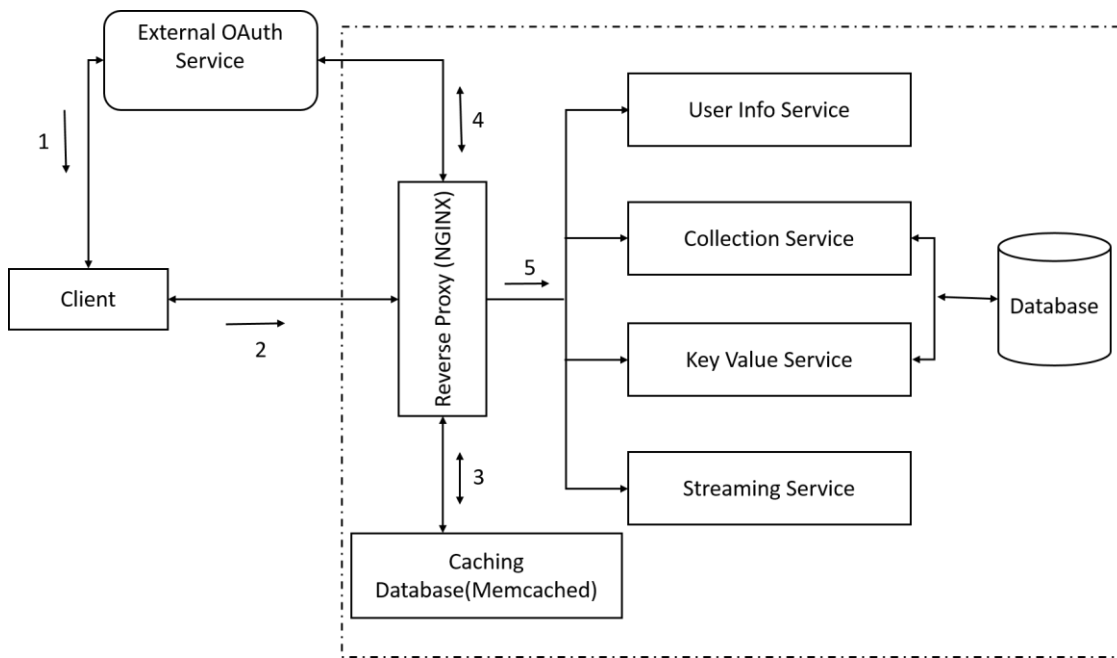


Figure 17 Implementation of Back-End 2

1. Client get an access token after passing his username and password
2. Pass this access token to microservice system
3. Reverse Proxy check whether user has already logged in by checking the cache database
4. If user is not already logged in, then reverse proxy will verify user details using external OAuth server
5. After validating user, his information will pass to the internal service

4.4.1 Reverse Proxy

In the second backend a reverse proxy used for user authentication. Here the reverse proxy used to do couple of things like token caching, token translation and various other things. Furthermore, all the requests to the internal services are going through the proxy server. This means there will be a high overload to the reverse proxy. So when selecting a reverse proxy, we should find a reverse proxy with good performance. Here we will consider Apache and NGINX reverse proxy servers; these are two common servers using as reverse proxies.

NGINX is event based server and Apache is process based server. So Apache server has to create separate threads for each of the user connection. But NGINX is using an asynchronous mechanism to handle user requests. NGINX will not create multiple threads for each of the user connection and all the processing is done in a simple loop over a queue in one thread. This unique architecture of NGINX server allows it to run with a very small memory footprint and to handle high number of user requests. So based on the above reasons NGINX is used as the reverse proxy. NGINX proxy will have to do several things such as token translation, token validation, forwarding request and caching tokens. To do all these things we have to implement a custom logic in reverse proxy server. NGINX support both C and LUA programming languages. So we can use either of these two languages to implement the whole process in reverse proxy.

4.5 Data Caching

A caching mechanism used in both of the implemented back-ends to improve performance. Because we don't want to store user information permanently, in memory database is the most suitable option. There are several in memory databases such as Redis and Memcached. Among these systems, Memcached is used as the caching database. The main reason for this decision is the simplicity of the Memcached database. Memcached is a key-value database, all the data stored as key value pairs. In the first implementation, microservices are directly connecting with the caching database and in the next implementation, reverse proxy connecting with the caching database. One important difference about the caching database in these architectures is, in first implementation caching database lives in a separate container, because every services have to access it. But in the second implementation caching database and the reverse proxy lives in the same container.

In this system, there are several key value pairs that we should store in the cache. First we should store the access_token that the user has passed and the access_token fetched from the authentication server. Additionally, access_token, id_token key value pair and access_token, user_info JWT pair should store. So when user accessing the service, reverse proxy or microservices will check the cache database with access_token. In Memcached all the values are store as key and a value pairs, and the key should be a unique value. But when we trying to save above mentioned values, we get three values for the same key. To avoid this problem a string append to key appended, "#TableName". The TableName value can be, USER_INFO, ACCESS_TOKEN or ID_TOKEN.

4.6 Database Usage

Some of the services in this system store and fetch user specific data. For example, both of the collection and key-value store service are saving collection and wallet information with userid. This functionality is implemented in both of the back-ends. For this purpose, a separate database has used. In this implementation MongoDB is used for the backend database. MongoDB database instance is running in a separate container.

In MongoDB data saved as documents. After saving the data, services must be able to take the data and to show them to specific users. To identify the owner of data, services will store data with userid or user email address. So when retrieving the data, service can check currently logged in user and according to that service can decide what information should include into the response. Three collections created in the database collection, wallet and user. Collection holds information about available collections and the user, user collection stores about authenticated users and the wallet holds key value pairs. Following examples show how a collection stored in the database.

User collection information:

```
{"_id": ObjectId("58a31776af9d2f1333970ddf"), "email": "johnwick@gmail.com"}
```

Collection holds another collection named Country, created by user with email address "johnwick@gmail.com"

```
{"_id": ObjectId("58bdad89a09870b208b6383e"), "userid": "58a31776af9d2f1333970ddf", "name": "Country"}
```

4.7 Implementation of reverse proxy to microservice communication

As mentioned in the chapter 3, user tokens are signed using a public/private key pair by reverse proxy before sending to microservices. Then microservices will validate the tokens using public key of the reverse proxy. A public key store will be used to handle key transferring process. A public key store technology called Vault is used to store public keys. This service will run in a separate container. Microservices will make a REST based system call to this service and request the public key. And the vault based microservice system will send public key as the response to the request. The format of a request is,

Hostname:port/service/getkey?containerid="containerid"

4.8 Implementation of logout service

As mentioned in system designing section, system should terminate user session in two situations. In first case, a user may call logout service and in the second case user's access token may have expired and system automatically terminate the user account. And there are two tasks that the system should do as a result of user logout. At first, the user details in the Memcached database should invalidate, secondly, each of the microservices should inform about the logout action. To invalidate user data, reverse proxy update all three database values associated with logged out user with an empty string.

After deleting user information, reverse proxy should inform this event to other microservices. RabbitMQ, publisher/subscriber mechanism used for this purpose. Reverse proxy will pass an event to RabbitMQ server and RabbitMQ server will send this message to subscribed microservices. In our backend system, microservices like "userinfo" is not interesting about the user logout event, so it may not subscribe for this event and it will not receive user logout event. This event is important for streaming service, which will terminate streaming process to the user upon receiving this event. Figure 18 shows system architecture used to process user logout.

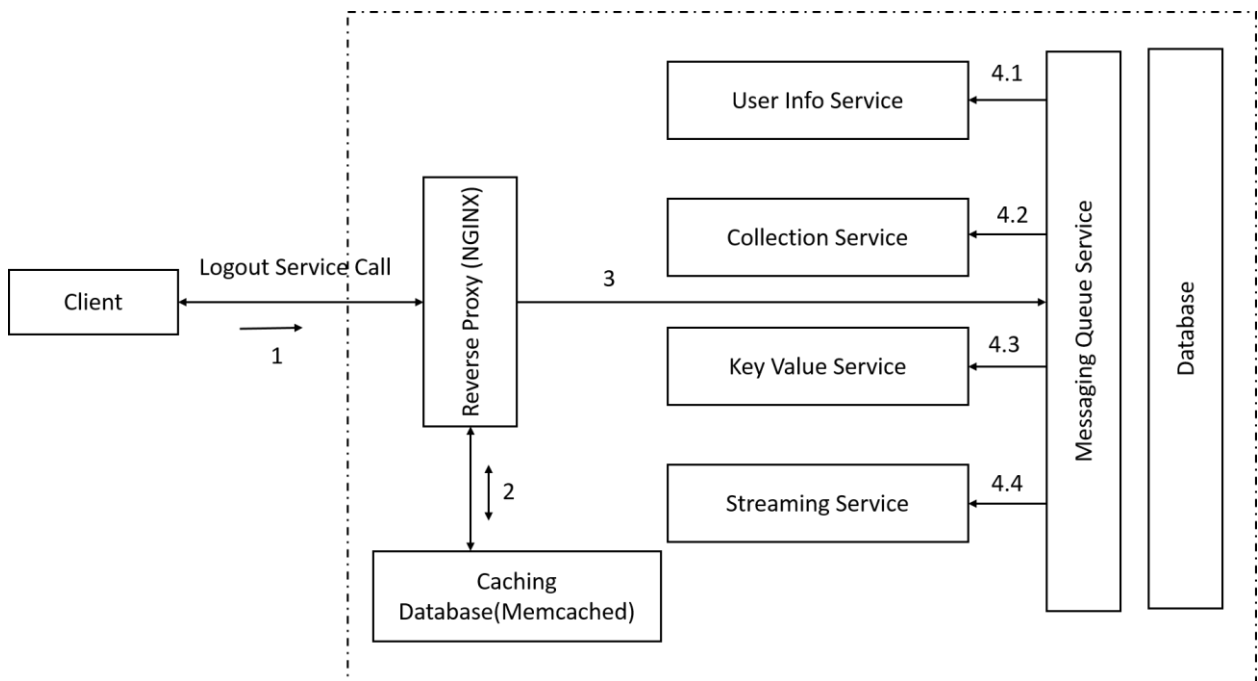


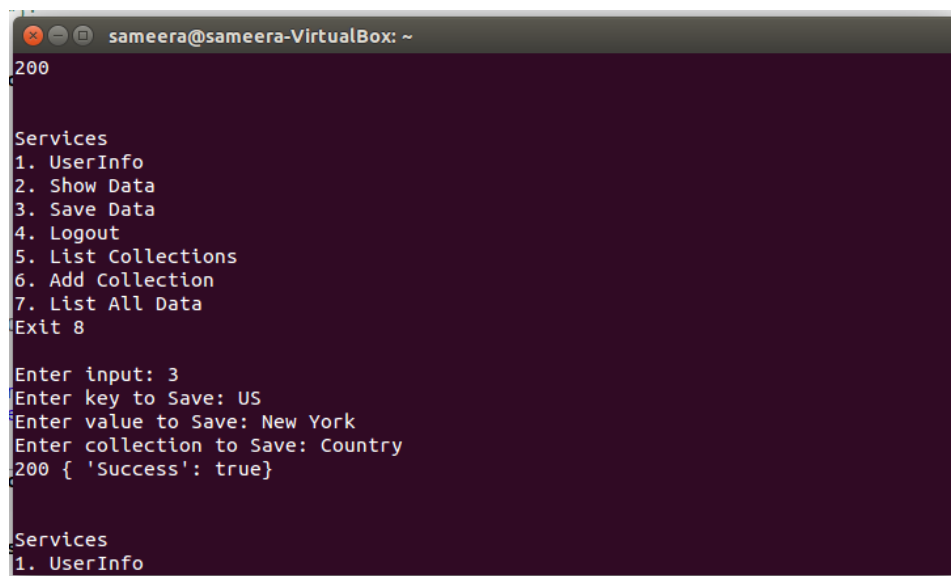
Figure 18 Implementation of logout mechanism

1. Client invoke logout service call
2. Getting user information from cache database
3. Pass logout event to Messaging Queue Service with logged out Userinfo
4. Pass event to registered services with user info

Finally, there is another scenario where we have to terminate user sessions. User sessions should terminate on token expiration; tokens contain the expire time as a claim value. So after a specific time an event should pass to all microservices informing the termination of the user session. This process is handled by RabbitMQ using scheduled process [13]. So after the token expiration, RabbitMQ will pass a set of event to each of the microservices.

4.9 Client Implementation

As shown before, two back-ends developed to check performance and feasibility of each of the systems. To test this systems, two client applications developed. First client developed as a command line interface using python. Then the second client developed as a GUI desktop app using Elctron. Each of these two clients can login to the system, logout the system and invoke each of the microservices in the backend. Before login to the microservice system, clients should also login to the authentication server, after login to the authentication server client will get an access token and then it should pass to the backend microservice system. Then client should perform subsequent service invocations using the correct parameters as required for the services. Figure 19, 20 and 21 shows the interfaces of clients,



```
sameera@sameera-VirtualBox: ~
200

Services
1. UserInfo
2. Show Data
3. Save Data
4. Logout
5. List Collections
6. Add Collection
7. List All Data
Exit 8

Enter input: 3
Enter key to Save: US
Enter value to Save: New York
Enter collection to Save: Country
200 { 'Success': true}

Services
1. UserInfo
```

Figure 19 Command line client

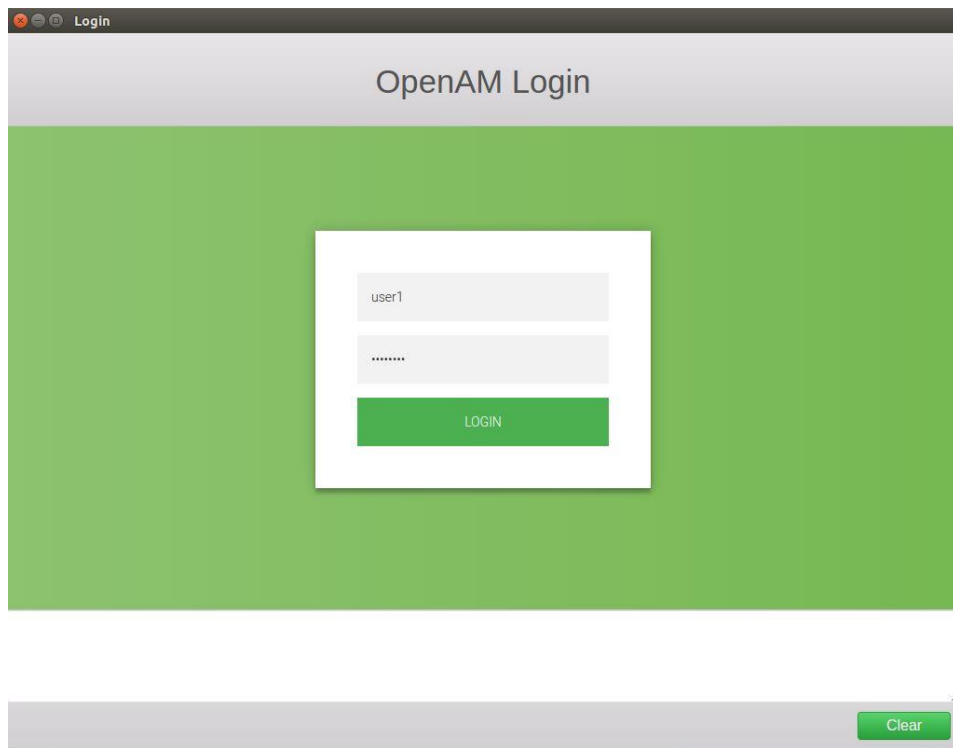


Figure 20 GUI client user login

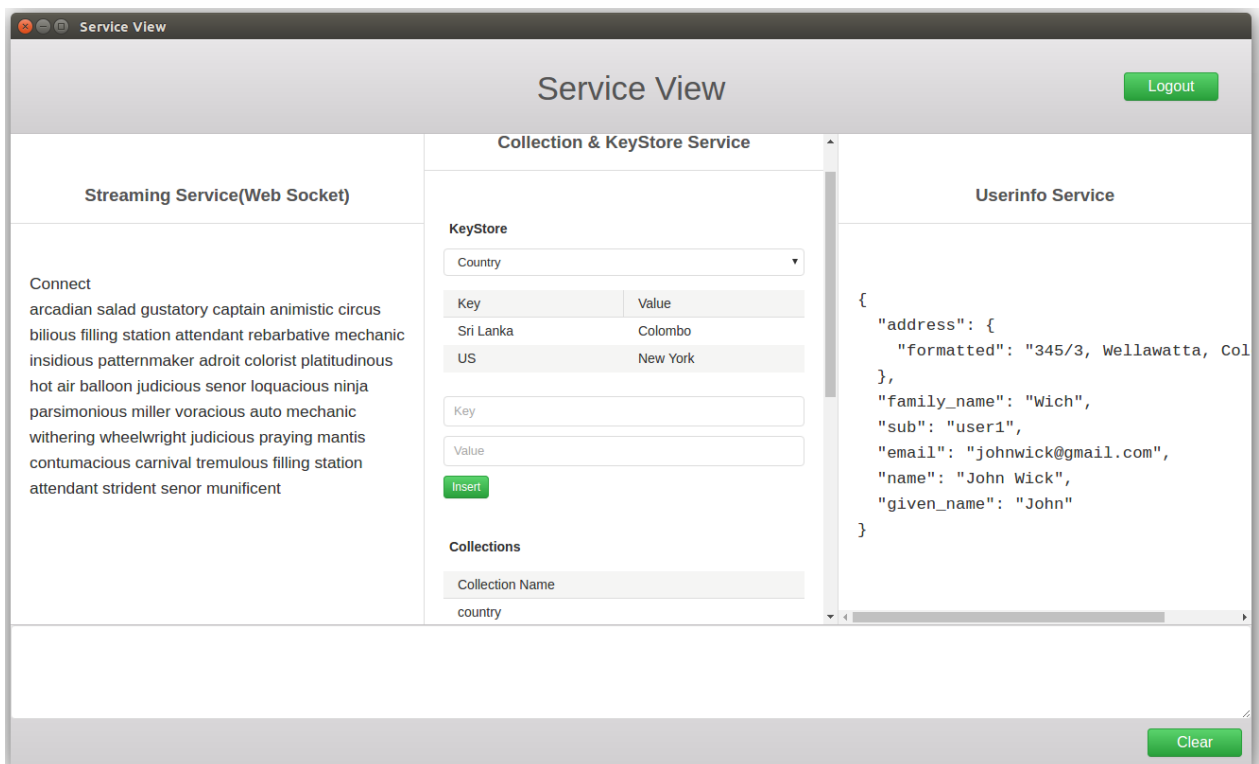


Figure 21 GUI Client Main Window

4.10 Technologies used

Several technologies used in the implementation of this system. Basically Docker used as the platform for the microservices and top of this, microservices built using different languages. All the technologies used in the system and the purpose of using them are lists in following table 15.

Technology	Use of the technology
Docker	Docker is an open platform to develop applications in containers. In this application, all the microservices, reverse proxy and in some cases the databases are deployed in separate containers. In docker we can create separate networks, so containers within the same network can communicate with each other. Two docker networks created for first and second back-ends.
Node.js, PHP	These languages used to develop backend microservices. In first backend, authentication procedures programmed using these languages.
NGINX, LUA	NGINX is used as the reverse proxy in the second architecture to token translation, token caching and to invoke services. LUA used as the programming language to code inside the NGINX server.
Memcached, MongoDB	These technologies used as databases. Memcached used as the caching database and the MongoDB used as the database to store user information.
OpenAM	OpenAM used as the OAuth/OpenID client.
RabbitMQ	RabbitMQ used as the messaging queue service. This used to handle user logout process.
Vault	Used as a public key store
Python	The command line client developed to test this system and it programmed using Python. Other than that in the performance tests performed using python.
Electron, Photon	The desktop based GUI client developed using Electron, which is a HTML, CSS and Javascript based GUI environment. Photon used as a CSS library.

Table 15 Technologies used in this system

4.11 Summary of Chapter

This chapter contains implementation details of the system design mentioned in chapter 3. Two back-ends developed on a Docker environment to handle user authentication. Several example microservices including a stream service developed in each of these back-ends test system functionality. NGINX reverse proxy used for user authentication in proxy based system and used a Memcached database to improve system performance. A RabbitMQ based publish-subscribe service used to handle session termination, the asynchronous nature of this system will reduce the workload on reverse proxy. Furthermore, two client applications developed to test the system, one of them is a python based command line client and the other application is an Electron based GUI client.

Chapter 5: Results & Evaluation

This chapter discuss about the final outcome of the system and overall successfulness of the system. The project focused to find answers for how to authenticate users in microservices and how to manage user sessions in microservices. User authentication process designed and implemented using two back-ends. These two back-ends can use for benchmarking and later these results can compare in term of security and performance. Furthermore, two client prototypes implemented to test system functionality and these prototypes can also use for system evaluation. A combination of all these things will be used in this chapter for evaluation.

5.1 Performance Benchmark Results

Basic functionalities of the system implemented using two back-ends. Different type of tests performed to evaluate the strength of the system. In here first I will present the performance related tests and then I will present security related tests. Performance related tests conducted for both of the systems in different granularity levels, component level and end to end system level. Most of the performance tests can be considered as capacity tests. In capacity tests, capacity increased by increasing the number of concurrent user requests.

Python based test applications used for benchmarking and these scripts used to test both of microservice back-ends. In the benchmarking, test scripts acted as clients for Docker based microservice system. Same python scripts used to test both of the microservice back-ends, the only change made to scripts was changing IP addresses. In end to end tests, time measured as the interval between sending a request and receiving a response. For capacity tests, the number of concurrent user requests increased by using multiple threads in python.

In component level system testing, reverse proxy and microservices used as components. In this case, test scripts programmed using the same language as the component and the time for tests measured as the interval between arrival of a user requests and sending response. For example, in microservice testing, the time interval measured as the difference of the request arrival time and the response sending time. user_info microservice used for all of the following benchmarking tests. It choose, because it's the microservice with minimum overhead to the system. In all of the following benchmarking results, a number of concurrent users sent multiple requests, each user sent 10 requests sequentially. So the time per request measured as the,

Time per request = (Total time each user taken to complete test) / 10 * Number of concurrent users

In all of the following graphs time is measured in milliseconds. When performing these tests, two computers used to run client test scripts and a separate computer used to run Docker based microservice system. These two computers connected to a local area network to communicate. Ubuntu 16.04 based Docker images used for microservices. System specification of these two systems are lists in table 16,

System specification of microservice system machine	System specification of client machines
CPU - Intel Core i5-2430M	CPU - Intel Core i5-2430M, Intel Core i3 2330M
RAM – 8GB DDR3	RAM – 4GB DDR3, 4GB DDR3
Operating system - Ubuntu 16.04 LTS	Operating system - Ubuntu 16.04 LTS, Ubuntu 16.04 LTS
Hard drive – 500GB HDD	Hard drive – 500GB HDD, 500GB

Table 16 System specification

5.1.1 End to end level performance evaluation

In the end to end level performance evaluation, time per requests measured as the time difference between sending a request from the client and receiving response from the microservice system. Figure 22 shows the results of Back-End 1 and the figure 23 shows the results of Back End2. One important thing to note is, these tests were performed separately for the first request and for requests after the first request. The reason is on the first request microservice system has to connect with the authentication service to validate user and on subsequent requests microservice system only using the user information stored in the cache. Following graphs shows the benchmarking results of time per request vs number of concurrent users for the user authentication in back-end 1 and back-end 2.

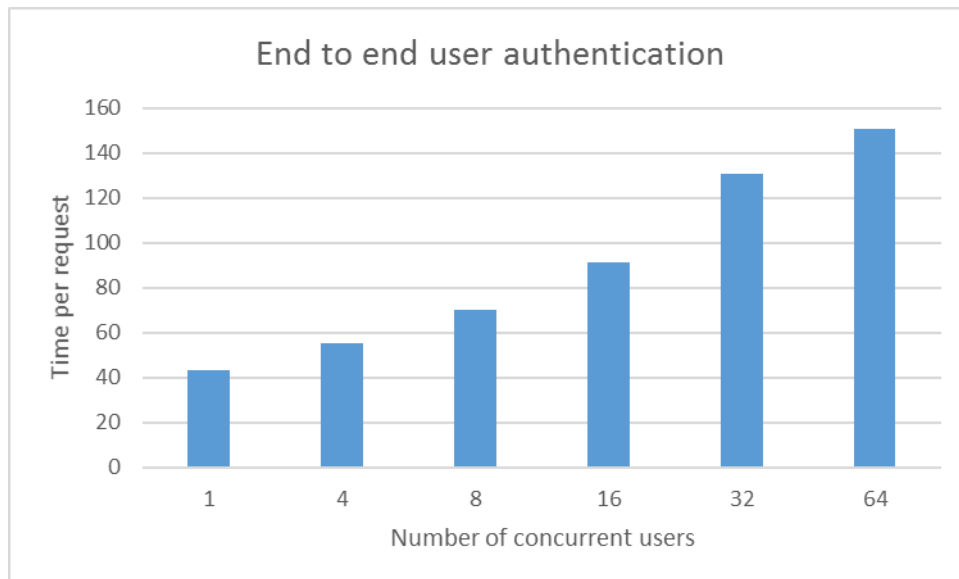


Figure 22 Performance evaluation user authentication back-end 1

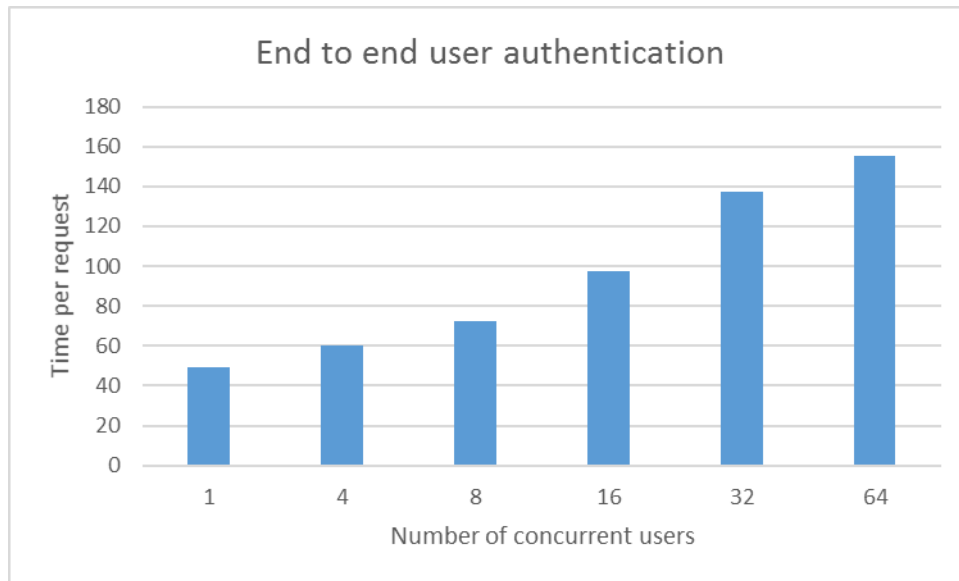


Figure 23 Performance evaluation user authentication back-end 2

Figure 24 and 25 shows the benchmarking results of time per request vs number of concurrent users for the requests after the user authentication in back-end 1 and back-end 2. Calculations were done as same as before,

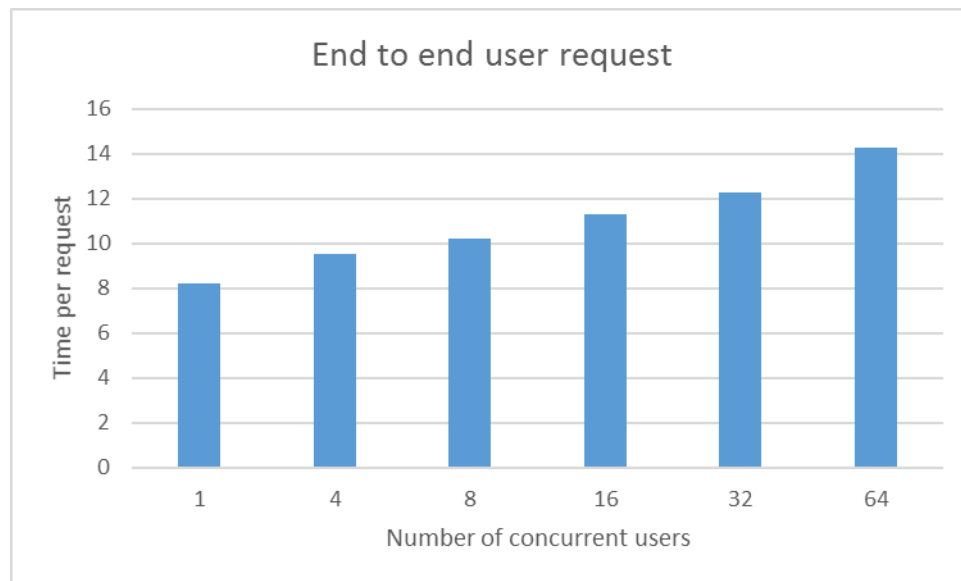


Figure 24 Performance evaluation for end to end request and response back-end 1

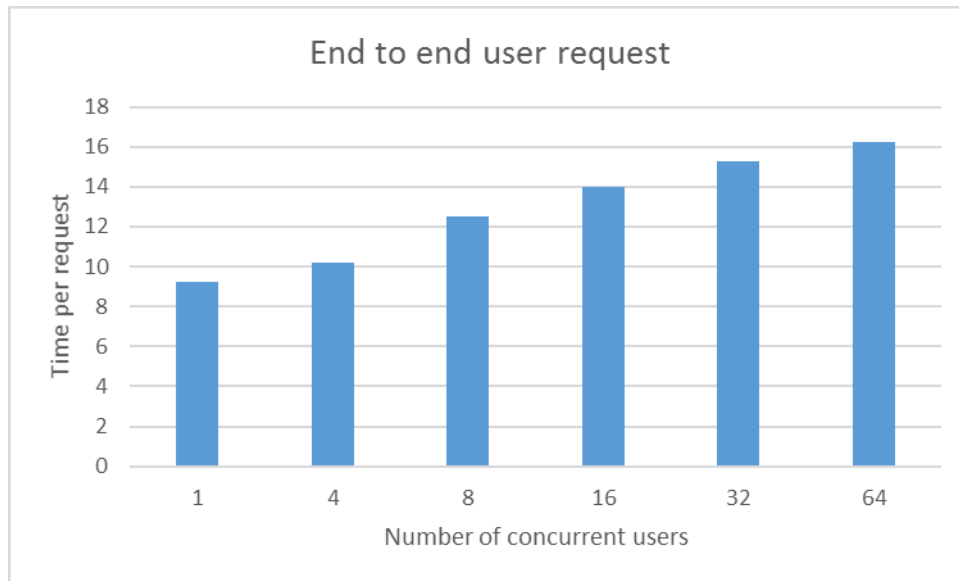


Figure 25 Performance evaluation for end to end request and response back-end 2

5.1.2 Component level performance evaluation

In component level performance evaluation, performance measured in each of the significant system components. Reverse proxy in Back-end 1 and reverse proxy to microservice communication is identified as the most important points to measure the performance. These measures performed only for the Back-End 2. Figure 26 shows the time taken by the reverse proxy for user authentication. Figure 27, benchmarking result shows the time taken by a microservice to respond back to reverse proxy. So this measure includes the time taken by service to verify token signature and to process information. Measured the time required for a microservice to send a request to another service and to receive it in other end.

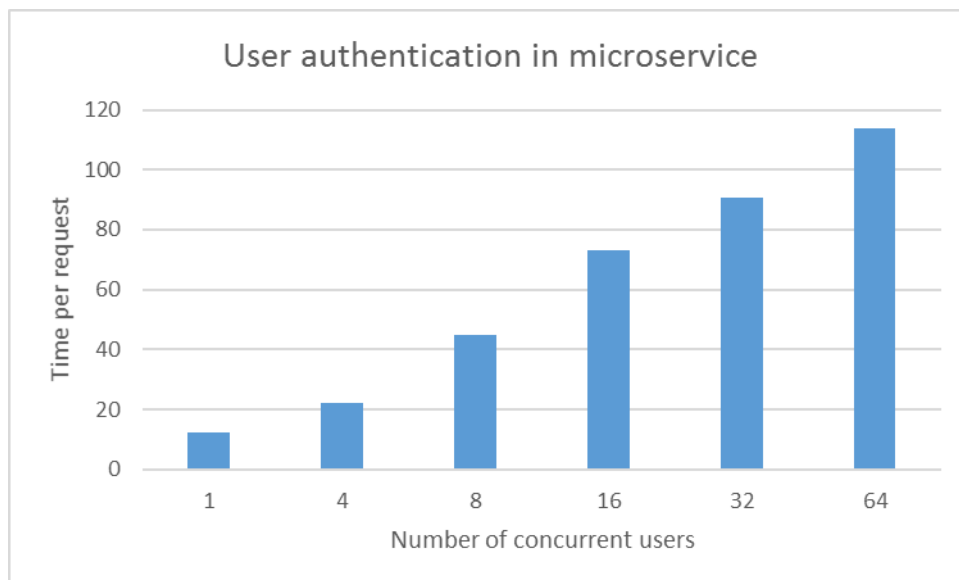


Figure 26 User authentication in reverse proxy

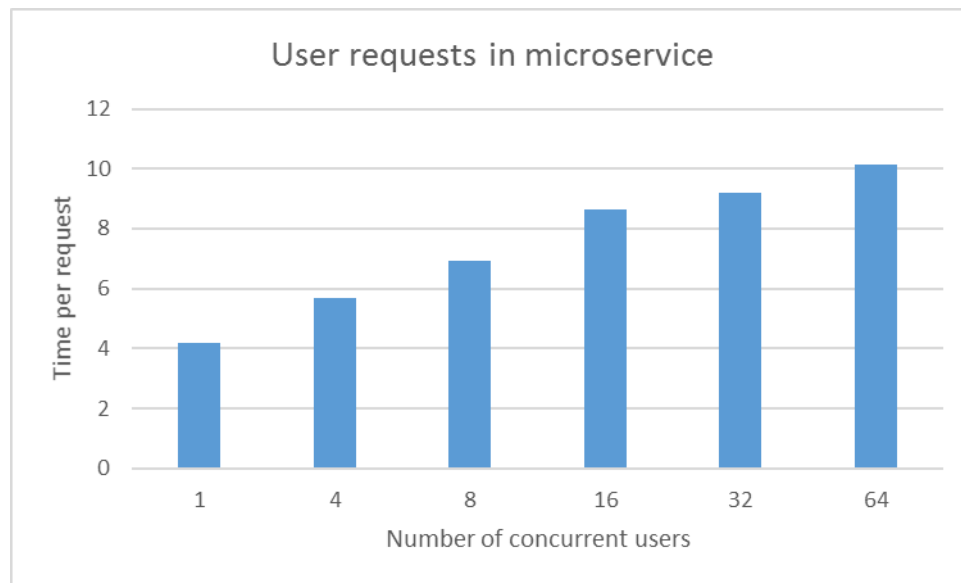


Figure 27 Time for user requests in microservice

5.1.3 Performance Evaluation

A number of benchmarking tests conducted and its result is shown in figure 22 to figure 27. Figure 22 to Figure 25 shows the time taken to user authentication and time taken to send requests after the user authenticated by Back-End1 and Back-End2. According to these graphs Back-End2 shows less performance compared to Back-End1. The time taken for a request getting increase in both of these systems as increasing the number of concurrent users. But none of these graphs show exponential growth in time as increasing the number of concurrent users.

5.2 Security test results

Several security tests performed to evaluate security of the system. Initial tests aimed to test system capability to face Dos Attacks, Reply attacks and Man in the middle attacks. Following sections contains details about the setup used to do these tests and results of each of those tests. Finally, a number of tests performed to validate the system against OWSAP security standards. A conclusion about the results of these security tests is made in chapter 6.

5.2.1 Dos Attacks

There are two places where system communicates with public internet. One place is between client and NGINX proxy, other place is between NGINX proxy server and authentication server. Among these endpoints, authentication server to NGINX proxy end point will not be able to use for DoS attacks. Because request for the authentication server always initiate by the NGINX reverse proxy, attacker will not be able to use this end point. Here evaluated system security for DoS attacks which initiated from client to NGINX proxy server. Here I make the assumption that the DoS attacker will not be able to get a valid OAuth access tokens. So a set of requests with invalid access tokens which initiated by the attacker is sent to the microservice system, and a valid user requests which initiated by a real user is passed to the backend services. Then measured the time taken to complete the real user request.

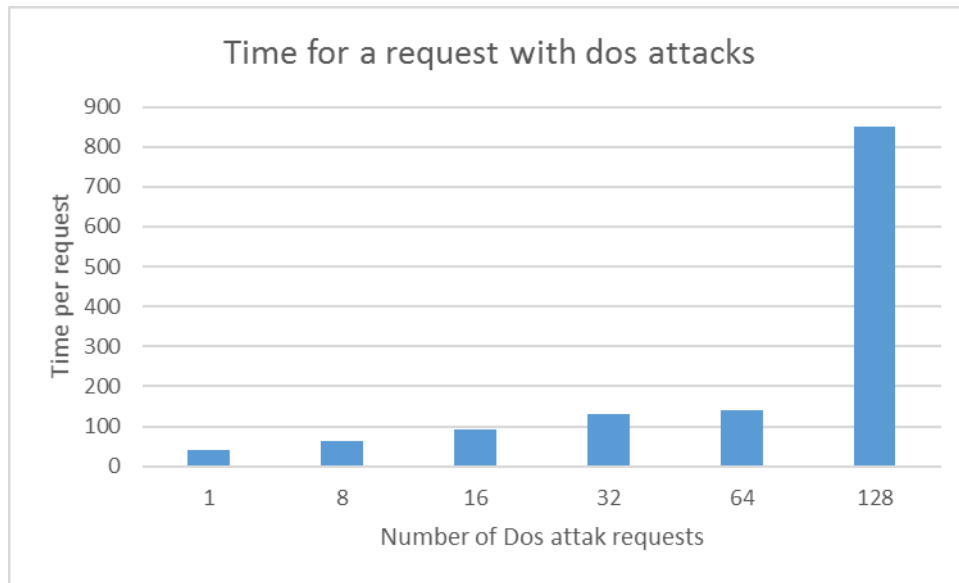


Figure 28 Time for user request with dos attack

5.2.2 Reverse Proxy to Service Communication

In this system various measures have taken to prevent man in the middle attacks. So here system tested with several man in the middle attacks. A reply attack performed after capturing a token and resending the same token and a token passed to the microservice with wrong signature. These tests made on collection service and wallet service. Results of these tests lists in table 17,

Test	Method	Result
Reply Attack	Capturing the same valid user access token and pass it to a microservice	{ success: false, error: Failed to authenticate }
Pass invalid token	Create a token with an email address of a registered user and pass it to microservices	{ success: false, error: Failed to authenticate }

Table 17 Results of security tests

5.2.3 Security result evaluation

To evaluate system security several attacks were made, DoS attacks, Reply attacks and passing invalid tokens as a man in the middle of the system. The DoS attack performed here is not a full scale attack, but as results shows we can predict that an increased number of invalid requests will degrade system performance significantly. Even though, we passed invalid tokens to the reverse proxy, still reverse proxy should call authentication server to validate them, this may have cause to lower the system performance considerably. System showed considerable good resistance for both of reply attacks and man in the middle attacks. Because tokens are validating using nonce value,

system could identify reply attacks and to not perform any actions. In the case of passing invalid user tokens, system could identify the tokens as an invalid after verifying the digital signature.

5.2.4 Security evaluation against OWSAP top ten vulnerabilities

Front-end and the back-end of the system tested against OWSAP top ten vulnerabilities to verify the security strength. Following are the system responses for these vulnerabilities,

A1 Injection – Backend server accepts requests which coming from the client and then it redirect the request to the correct backend service. In this process client pass parameters to the backend service in the URL.

For e.g.:

hostname:port/service/country/operation/add?collection=Country&key=India&value=Delhi

Proxy based backend service only accepts service and operation parameters and all other parameters are directly passing to the backend system. This makes injections are not possible due to the fact proxy server only using the service name and operation name to decide the backend service.

A2 - Broken authentication and session management

In this case two security tests were performed to measure system security strength. These test results are lists in the table 18.

Test	Response – HTTP Status code
A logged in user's access token was saved. Then logged out the user and attempted to access a service by passing the logged-out user session id.	401 Unauthorized
A logged in user's access token saved and the token passed using another browser and attempted to access the services.	401 Unauthorized

Table 18 Broken authentication and session management tests

A3 Cross site scripting (XSS) - Backend proxy service is only using the operation and service parameters to access the services, so it is impossible to make XSS based attack.

A4 Insecure Direct Object References - None of the backend objects are exposed except the service names. Invalid service names and operation names are passed to the backend but the result was 404 Not found in all those attempts.

A5 Security Misconfiguration - To test security misconfigurations invalid REST requests methods types are passed. Backend service only supports the POST, HEAD and GET requests. The response for the PUT and DELETE requests to the proxy server was 405 Method Not Allowed.

A6 Sensitive Data Exposure - In this case user password is considered as a sensitive data. None of the user passwords are stored in the proxy server, all of them are stored in the OAuth server and the connection between OAuth server and the client application is secured using a HTTPS connection. Because of these reasons it's impossible for attacker to get user passwords from the backend server.

A7 Missing Function Level Access Control - Tests made for "Broken authentication and session management" can also be used in here to verify that not only the client, but the backend services are also validating the user requests.

A8 Cross Site Request Forgery - A request made to the backend server using another web page. And the server responds with 401 Unauthorized request. In this case backend server, has checked the Origin Header, the request rejected because this is not the same address as the server.

A9 Using Components with Known Vulnerabilities - Most up to date system components were used in the system to reduce the known vulnerabilities in the system components. Latest NGINX server version 1.10.3 is used as the backend service.

A10 Unvalidated Redirects and Forwards – Proxy server doesn't perform any client redirections, so this vulnerability will not be able to use to exploit the backend server.

5.3 Functionality evaluation

Functionality evaluation tests will be done using two client prototypes developed as a part of this research, a command line based prototype and a GUI based prototype. These systems are shown in Figure 19, 20 and 21. Both of these systems used to evaluate the system functionality. Table 19 lists some of the tests performed and the result of those tests.

Test Description	Client	Expectation of the test	Result
Basic functionalities – User info service, wallet service and collection service	Command Line/GUI	Success on each functionality	Pass
Save data to collections created by other users	Command Line	Failed to save data	Pass
Login two users same time and create collections	Command Line	Create collections correctly to each user	Pass
Login two users and check web socket output	GUI	Socket output should distinct for each user	Pass
Web socket output should stop on user logout	GUI	User logout event pass to microservices properly	Pass

Login to backend and keep session idle till token expire and try to use system	Command Line	User session should be terminated when sending requests after token expiration	Pass
--	--------------	--	------

Table 19 Functionality evaluation

5.4 Overall system evaluation

Objectives of this research is to find a secure and efficient way to authenticate users and handle sessions in microservice systems. Two back-ends designed and implemented to measure these properties of the system. Both of these two systems have advantages and disadvantages. In first backend user authentication performed at microservice level, this back-end showed a slight performance improvement compared to the other system. One of the biggest advantages of this back-end comes with its distributed nature. Even if a single component failed, the entire system will function properly. But maintaining this backend was rather difficult. In the implementation chapter, several microservices implemented using several programming languages. So it had to write the same authentication logic for each of these microservices separately, and when it need to change the authentication logic, all microservice had to update with the change. Another disadvantage is, each services has separate IP addresses, so client should know IP addresses of every microservices.

In the second back-end user authentication performed by a reverse proxy. This system showed slightly less performance than the other back-end. One of the biggest disadvantage of this system is single point of failure, if reverse proxy failed then entire system will fail. On other hand, this system was easy to maintain. Complete authentication logic resided in the reverse proxy, so if it need to make a change, then we will have to make it only in a once place. Another advantage of this system is, all the services can invoke using a single IP address, and then the reverse proxy will decide the microservice it should invoke. Among these two back-ends the second back-end will be more practical even if it shows less performance. The security measures taken to avoid reply attacks and the man in the middle attacks are successful and the system could successfully block those attacks. Overall system functionality was better, functionalities like microservice operations, user logout mechanism and maintaining user session is functioning well.

5.5 Summary of chapter

This chapter contains information about various tests performed for the microservice system, test result and an evaluation about the results. Tests performed in this chapter can be divided into three categories as performance related tests, security related tests and functionality related tests. Performance related tests were done for two back-ends, and the back-end which perform user authentication at microservice level showed slightly better performance than the other. In security tests reply attacks, DoS attacks and man in the middle attacks were initiated and the system was secure for both of reply and man in the middle attacks. But the system showed low performance when DoS attack introduced. On overall, system shows good performance and even though the backend which authenticate users in reverse proxy shows less performance it has couple of advantages over the other system.

Chapter 6: Conclusion & Future Work

Microservice architecture introduced as an alternative to the monolith architecture to solve many problems such as application deployment and fault tolerance. There are lot of new security problems that developers have to solve with this new architecture. This research aimed to solve two major security problems in microservices. First problem is, how to authenticate users and distribute user information among services and then how to create user sessions and manage them. Access delegation services is used to perform user authentication. Then two back-ends developed to perform user authentication. One of these backend uses a user authentication architecture similar to what we use in monolithic systems and other one is different from this. In the first backend each of the micro services itself performed the user authentication and in the second back-end this process handled by a reverse proxy server.

One of the focus of designing this system was performance. A caching database is used to improve performance in both back-ends. And importantly some services made as asynchronous services to reduce the workload on the reverse proxy. Another focus on designing the system is security. To improve system security two authentication tokens types used in outside and inside of the system. Tokens which received from authentication server validated using various claims and these tokens passed with a new digital signature to other microservices. Other than user authentication, another area focused in the research problem is session handling. In this system tokens used to handle user sessions and this helped to get rid of using the cookies to handle user sessions. And on the event of user logout asynchronous mechanism used to terminate user sessions.

Various tests are performed to evaluate the system, including performance benchmarking, security evaluation tests and functionality evaluation tests. Among these tests, performance benchmarking conducted on both of the back-ends. This result shows slight performance increase in the backend, which authenticate users at microservice level. Furthermore, the time taken for requests fall down considerably after user logged in to the system. Additionally, combined result of component level tests and end to end tests shows that a significant amount of time is taken by authentication server.

Several security tests were performed to evaluate system security. These tests included DoS attacks, reply attacks and man in the middle attacks. The system was strong enough to face both of reply and man in the middle attacks. But for DoS attacks system showed a considerable degrade in performance. Functionality evaluation tests performed using two client applications, command line based and GUI based. These two systems passed all the tests in functionality evaluation. Finally system security tested against the top 10 vulnerabilities listed under OWSAP top 10. System passed all the tests performed against OWSAP listed vulnerabilities, except the Injection, Cross site scripting (XSS) and Unvalidated Redirects and Forwards. Attacks for these three vulnerabilities did not executed, because current system architecture and functionalities doesn't allow to perform such attacks.

Even though the first back-end which handled user authentication in microservice level, showed good performance, it's much more difficult to maintain than the other system. In the implementation section, multiple micro-services implemented using different programming languages. So it had to rewrite same authentication logic using programming languages of each of the micro-services. It was difficult to maintain and authentication component even in micro-services programmed with same language. In reverse proxy based microservice system also have some token validation operations that should do by microservices. But that coding part is

comparatively small in size and based on computer standard like JWT token validation. So those functionalities can implement easily using standard libraries.

Even though proxy based system showed considerable advantage in maintaining the system, its biggest limitation is single point of failure. But the first backend which authenticated users at microservice level is more tolerant to single point of failure issues. One way to solve this problem is using multiple reverse proxies and a load balancer. But this will again introduce single point of failure at the load balancer, but the expectation here is to handle as small as possible logic in load balancer and it will simply redirect. Another well-known way is to run multiple slaves, all of the servers will exchange heart beat and if one system failed another slave will elect as master. [11] [12]

Session handling mechanism was rather easy to handle than traditional cookies based system. But if microservices wants to create new information related to the user, then they will have to store information locally. It was easy to handle user sessions, because system used token based sessions rather than using cookies based systems. To terminate sessions asynchronous system calls used with a messaging queues and it was really efficient. Otherwise proxy server will have to pass each of these events to the microservices and it will increase the overhead on the server.

As already mentioned and evaluated in chapter 5, one of the crucial security problem of this system is inability to face DoS attacks. It may need to implement an intrusion detection system or NGINX itself has a module called `ngx_http_status_module` to face DoS attacks. Other than that IP blocking and continuous system monitoring to identify susceptible request patterns will help to mitigate this problem. It's worth mentioning two other problems faced when developing the system functionality, one of them is the same port in reverse proxy couldn't use for web sockets and other services, this is a limitation in NGINX reverse proxy. Another problem is, it was very hard to maintain IP addresses of each microservice. Microservices communicate with each other using IP addresses, but lack of centralized mechanism to know other service addresses was a time-consuming process in system development.

Finally, as a summary. the reverse proxy based authentication system shows less performance than microservice based authentication system. But the increased maintainability and easy deployment of proxy based system make it a good candidate for user authentication in microservices based systems. Various mechanisms like using caching database, validating tokens, use of different token types and signing tokens can be used to improve system performance and security of the system. Even though the proxy based system shows some advantages, it has some limitations like single point of failure and inability to face DoS based attacks.

6.1 Future work

- Authentication server always pass tokens with an its own signature. To validate this token public key of the authentication server is stored locally in reverse proxy and in microservices (In microservice based system). Instead of this we can get this certificate automatically on start-up using technologies like JSON web key.
- In this system tokens signed by the reverse proxy before passing to the microservices. Then these tokens validated by getting a public key from a key store. This same mechanism can extend easily to secure service to service communication.

- Now sessions are destroyed only based on user request or expiration of authentication token. But this process can further improve by adding the functionality to terminate sessions if user idle for a specific time. This will allow to secure system more.
 - Tests performed in chapter 4 showed the system inability to face DoS attacks, this can be improved using intrusion detection system or using nginx reverse proxy modules like ngx_http_status_module.
 - Reverse proxy based system shows its inability to face single point of failures. One solution for this is to use multiple reverse proxies, a master of these proxies will handle communication and using heartbeat mechanism servers will monitor each other. If master fails, then a slave will be elected as a master. This mechanism will be ideal for this system.
- [11]

References

- [1] What are microservices? [Online]. <https://opensource.com/resources/what-are-microservices>
- [2]. The Ins and Outs of Token Based Authentication [Online]. <https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication>
- [3]. Cookies vs Tokens. Getting auth right with Angular.js [Online]. <https://auth0.com/blog/angularjs-authentication-with-cookies-vs-token/>
- [4] What is Microservices Architecture? [Online]. <https://smartbear.com/learn/api-design/what-are-microservices/>
- [5]. RFC 6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage [Online]. Available: <https://tools.ietf.org/html/rfc6750>
- [6]. Building Microservices, 1st ed., Sam Newman, 2015
- [7]. Security-as-a-Service for Microservices-Based Cloud Applications - Yuqiong Sun, Susanta Nanda and Trent Jaeger
- [8]. Microservices a definition of this new architectural term - Martin Fowler
- [9]. Microservice Architecture Aligning Principles, Practices, and Culture 1st edition Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen, 2016
- [10] Kerberos authentication over the public internet [Online]. <http://security.stackexchange.com/questions/41803/kerberos-authentication-over-the-public-internet>
- [11] Bernard Rosset Architecting nginx for redundancy [Online]. <http://serverfault.com/questions/642589/architecting-nginx-for-redundancy>
- [12] Fast failover configuration with drbd and heartbeat partitions configuration [Online]. Available: <http://linuxmanage.com/fast-failover-configuration-with-drbd-and-heartbeat-on-debian-squeeze.html>
- [13] James Carr (2012 April 5) RabbitMQ: Scheduled Message Delivery [Online]. Available: <https://www.javacodegeeks.com/2012/04/rabbitmq-scheduled-message-delivery.html>
- [14]. Network security [Online]. Available: https://en.wikipedia.org/wiki/Network_security
- [15]. confidentiality, integrity, and availability (CIA triad) [Online]. Available: <http://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA>
- [16]. API Security: Deep Dive into OAuth and OpenID Connect [Online]. Available: <http://nordicapis.com/api-security-oauth-openid-connect-depth/>
- [17]. Prabath Siriwardena (2016 May 28) Building Microservices ~ Designing Fine-grained Systems
[Online]. Available: <https://medium.facilelogin.com/building-microservices-designing-fine-grained-systems-d37b57a97c4e#.f0e63nlve>
- [18]. JSON Web Token (JWT)
[Online]. Available: <https://self-issued.info/docs/draft-ietf-oauth-json-web-token.html>
- [19]. Chris Sevilleja (2015 Jan 21) The Ins and Outs of Token Based Authentication

- [Online]. Available: <https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication>
- [20]. Kerberos (protocol) [Online]. Available: [https://en.wikipedia.org/wiki/Kerberos_\(protocol\)](https://en.wikipedia.org/wiki/Kerberos_(protocol))
- [21]. Kerberos Explained [Online]. Available: <https://msdn.microsoft.com/en-us/library/bb742516.aspx>
- [22]. OWSAP Top 10 2013-Top 10 [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

