AUTHENTICATION AND DEVICE DISCOVERY AS A SERVICE FOR DEVELOPING LIQUID SOFTWARE APPLICATIONS

Pasindu Chandrasekara

(179309R)

Degree of Master of Science in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2019

AUTHENTICATION AND DEVICE DISCOVERY AS A SERVICE FOR DEVELOPING LIQUID SOFTWARE APPLICATIONS

Pasindu Chandrasekara

(179309R)

Thesis submitted in partial fulfillment of the requirements for the Degree of MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2019

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgment to any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in partial print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

---------- ----------

Signature Date

The above candidate has carried out research for the Masters thesis under my supervision.

Name of the Supervisor: Dr. Indika Perera

---------- ----------

Signature of the Supervisor Date

ABSTRACT

In this era where an average person consumes at least two computing devices, the applications that are developed for these devices should have the transformability among them. It not only ensures the users are not interrupted when switching between devices but also allows them to use the power of computing whenever, wherever. These kinds of applications with maximum transformability among computing devices are known by the term “Liquid Software”. Even though this concept of Liquid Software can be identified as a subsection of ubiquitous computing, it can prevail alone, whereas it is not the case for ubiquitous computing which cannot exist without liquid software. There are many architectural concerns that need to be addressed when developing liquid software applications. Application security and device discovery are two of the main architectural concerns of liquid software. Ensuring security while also maintaining liquidity in applications is a challenging task. In the application level granularity, device discovery when combined with proper authentication could provide a secure liquid experience to the users. But developing solutions while also addressing these concerns would require considerable amount of design and development effort. This research introduces a service model that could provide out-of-the-box authentication and device discovery features to assist development of liquid software applications. The authentication mechanism presented through this service model is mainly based on an authentication server and is also capable of providing service level security. The device discovery mechanism of the proposed service model is based on QR codes which could be controlled at the application level in order to be integrated with the authentication mechanism while hiding the complexity of device registration. The proposed Authentication and Device Discovery as a Service model could be either consumed by web applications to adhere to liquid concepts or extended by development teams in order to plug in their own services.

Keywords: liquid software, ubiquitous computing, device discovery, multiple device ownership

ACKNOWLEDGEMENT

This study would not have been possible without the guidance and the help of several individuals who extended their valuable assistance in the preparation and completion of this dissertation.

First and foremost, I'm grateful to my supervisor, Dr. Indika Perera, for his guidance, patience and providing me with an excellent atmosphere for the completion of this research work.

I thank all the teaching and administrative staff of University of Moratuwa, for their service and support.

Special thanks to all my colleagues whose commitments and support contributed to this project in numerous ways.

Finally, I would like to extend my gratitude to my family who encouraged me with all their heart and soul to make this study a success.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Liquid Software concept comes into the computing world alongside the concept of multiple device ownership. Today for anyone, it is the current norm to use at least two computing devices. The younger generation uses even more with the growing trend of smartwatches and other wearable devices. For these devices to coexist among each other they should have a common understanding, which is nearly impossible to maintain with the devices coming from different vendors with different capabilities. It is a well-known fact that it's the software that makes the communication happens between the hardware and the users. But, a piece of software which was developed for particular hardware or a software platform will not work on other platforms. The incompatibility of applications purchased from the apple app store on Android devices and vice versa provides evidence for this limitation. The differences in existing and upcoming platforms will prevail. Hence coming up with software which can adjust and coexist among these platforms will be the preferred and the novel approach. Software Architecture can be identified as the key component of this evolution, as it models possible solutions for this common problem.

Most of the existing cloud-based applications from vendors like Apple, Google and Microsoft can already run on both computers and smartphones with similar functionality. Hence it's evident that almost all of the application vendors are moving towards automatically synchronized devices. But this approach contains a considerable limitation since almost all of these existing cloud-based synchronized applications are vendor specific. Furthermore, the seamless transition of the application state among devices is yet to be discovered in a vendor-independent context. The Liquid Software Manifesto [1], states a set of requirements that should be met by a seamless multi-device ecosystem. Considering these set of requirements, it can be said that most of the cloud-based applications which are capable of staying synchronized with multiple devices are semi-liquid. In a true liquid environment, the user must be able to stay synchronized with all his/her devices with minimum configuration and maximum control.

While liquid software promises flexible application usage among multiple devices, the security of liquid applications plays a vital role. It is essential to have a robust authentication mechanism to identify trusted devices where liquid applications can operate in. It is essential that the users of these applications stay in full control of the device authentication process. While the authentication process should be as transparent as possible, it should also be robust. If a single authenticated device is compromised all application data of that user becomes compromised. Hence it is evident that security plays a major role in liquid software applications. Even though it is of utmost importance, the security of liquid software could be easily downplayed given its complexity while trying to achieve maximum flexibility.

The main objective of this research is to come up with a set of services which could provide out of the box features to assist liquid software development. The target audience of this research is mainly web application developers who wish to implement web applications that possess liquid properties while only focusing on their business logic. There are many architectural design decisions [3] that provide many alternative approaches during the initial design phase of software applications. While some of these architectural decisions are more meaningful when made considering the business requirement, there could be generic design decisions that could be transformed into services that could provide out of the box features to any software application. Authentication and device discovery are a couple of such decisions that we believe could be transformed into a service model that could be consumed by application developers who intend to include authentication and device discovery features to their applications with less effort. One of the main objectives of this research is to cover all points in the liquid software manifesto [1] that relate to device discovery and authentication. This provides the guarantee that if these service models produced as outcomes of this research are consumed in an application, they alone could make that application semi-liquid. If other aspects such as state replication are properly handled in an application while also consuming the proposed service models, such applications can be identified as pure liquid software applications according to the liquid software manifesto [1]. This research is mainly influenced by several previous works done under this domain [1-6], which could help identify further improvements and implementations which are possible.

1.1 Liquid Software

Liquid Software is the future of software applications. According to the current trend, it is possible that each and every one will have multiple computing devices controlling their daily life. Hence it is very important for all these devices to stay synced with each other. But the underlying hardware of these devices cannot collaborate. It's the software that let them collaborate. Such software which can pour from one device to another and perform the required operations is called Liquid Software. It's a major subsection of ubiquitous computing which gives the users the power of computing whenever, wherever. Today, several tech giants have already come up with marvelous creations with respect to multiple device ownership. Apple's Hand-off [10], Android Baton [11] from NextBit and Windows Continuum [12] are among the top competitors in the domain of Liquid Software. But their attempts are not purely liquid in nature as they are all native attempts. It's like a particular liquid can only be poured into a particular type of vessel. The most significant gap is the inability to maintain the state of the application among multiple devices and multiple ecosystems.

There are a few elements which make the liquid software different from other traditional solid software [2].

- Code Mobility - The ability of the code to execute in different environments. Strong mobility implies that the code can move along with the state while weak mobility means the code has to execute in a re-initialized manner.
- State Synchronization - The ability to maintain the state of the application among multiple devices. This will be discussed throughout the text as it is identified as one of the major limitations in the current cloud-based applications.
- Adaptation - Capability of an application to adapt to different contexts in different environments. Traditional software has to go through complex relocation, redeployment and reconfiguration of applications.

- Data and State - This refers to maintaining the application data across all devices in contrast to maintaining the application state. It's important to highlight the difference between the two when it comes to an application.

Each of these elements includes making proper design decisions that need to be taken in order to meet these requirements. Thus a set of generic design decisions that can be followed in order to fulfil these properties would be ideal when developing web applications that should be liquid in nature.

Designing Liquid software includes a lot of effort on user experience. Since switching among devices should be as casual as possible, it's a challenging task to provide a seamless user experience for the users. Different computing devices have different capabilities. While laptops have a keyboard for input, a mobile phone has touch-based gestures. And the screen size is also a variable that needs to be taken care of. Apart from all these, the usage pattern of the multiple devices is also a major attribute to be considered when it comes to designing Liquid Software Applications. Interactions with liquid software can be identified under three main categories [4].

1. Sequential usage
   A user runs the same application in multiple devices at different times.
2. Simultaneous usage
   A user runs the same application on multiple devices at the same time.
3. Collaborative usage
   Several users run the same application on multiple devices either sequentially or simultaneously.

All these usage scenarios can be seen in a number of systems that have emerged in the recent past with the power of cloud computing. Several popular names such as Google Drive, Apple's iCloud, Amazon Cloud Drive, DropBox and Microsoft SkyDrive has emerged as systems which support dynamic synchronization of data among different computing devices.

1.2 Problem

Since Liquid Software is going to be the future of software development, it's essential to find the means of addressing the main architectural concerns related to it. Even though considerable work has been done to bring together the potential design decisions [3] which are important when developing liquid software, they need to be put together in order to come up with generic solutions to these recurring concerns. Security is a major design aspect of liquid applications that can be easily overlooked. Applications that enable liquid nature should both be secure and easy to use. In the liquid software design space [3] the security aspect is not given much attention but it's a potential requirement to consider application security and authentication as important design decisions. Device discovery is also one of the key fundamental features of any liquid software application as it is the starting point for any kind of liquid experience. It is the ideal scenario to combine both authentication and device discovery together since it provides hassle-free roaming between devices while ensuring security.

If application developers themselves engage in the design and development of solutions to these concerns, it will directly affect their overall development effort resulting in late delivery. Therefore it is wise to look for *"out-of-the-box"* solutions to these problems. Popular service models such as Software as a Service, Platform as a Service and Infrastructure as a Service provide similar solutions to a different set of problems. But a service model that could provide solutions for authentication and device discovery concerns with regard to liquid software has not yet been emerged. Adopting such a service model would reduce the efforts of the development teams comprehensively while encouraging them towards developing liquid applications.

.

1.3 Motivation

As mentioned in the previous sections Liquid Software is a potential requirement when it comes to ubiquitous computing. In order to provide the power of computing to users whenever, wherever, we need to concentrate on the software that the devices consume, and their ability to transform. There are several limitations when switching among devices such as resource limitations, power limitations, the difference in input mechanisms and variable screen sizes. But application developers should not ideally worry about such limitations. In the current context, software development has its own set of requirements and constraints to be met. Therefore a potential mechanism should be available which can enforce the liquid properties to a software application. Furthermore, authentication is a major requirement when it comes to liquid applications. With the modern developments in technology, Authentication as a Service is a great resource for startups and small development teams to get their work up and running without worrying much about authentication. Authentication on its own cannot make a huge impact on the liquidity of applications. According to the liquid software manifesto [1] authentication, when combined with device discovery, covers at least half of the liquid properties that a common liquid software application should possess. Therefore when combined together authentication and device discovery could make a huge impact on the liquidity of applications. Considering the popularity of service models such as Software as a Service (SaaS), Backend as a Service (BaaS) and Platform as a Service (PaaS) it would be meaningful to have a service model that provides authentication and device discovery features. When features such as authentication and device discovery are readily available, application developers could consume them to kick start their development process. In such a scenario the application developers would start their progress not at 0% but from somewhere around 10 - 15% given that they have a mechanism to cover two of the most fundamental elements of liquid software.

1.4 Objectives

The main objective of this research is to come up with a service model that provides authentication and device discovery functionality to web applications. The authentication mechanism should also provide the means to secure the backend services of any application. Since it is the current norm to have independent microservices which provide modular functionality, it is required to have them secured as well. It is the expected behaviour that only users that are properly authenticated from within the application become eligible to consume the backend services. It is an objective of this research to adapt an authentication chain which is based on standard protocols such as OpenID Connect, OAuth 2.0 and SAML. The proposed solution would include modern features such as Single Sign-On to authenticated users among multiple devices.

1.4.1 Goals

- Develop a service model that provides authentication and device discovery as a service.
- Build a mechanism to impose security at the service level.
- Provide user management through the service.
- Develop a proof of concept application consuming Authentication and Device Discovery as a Service.

CHAPTER 2

LITERATURE REVIEW

The term *Liquid Software* first came into light through a technical report by Hartman, manner, Peterson and Proebsting in 1996[8]. Its primary intention was to enable the flexible use of network transported code on top of the Java Platform [7]. Their paper discusses use cases such as, the capacity to improve remote execution since it provides one site with the capacity to download those modules that are required to access the resources at another site. Remote execution also allows installation of software, diagnostics and maintenance irrespective of the physical distance among devices.

How do we identify Liquid Software could be a very interesting question. Since software is not a physical entity, it's harder to identify or categorize software unless there are predefined properties which assist in classification. The Liquid Software Manifesto [1] presents a clear set of requirements that should be met by an application for that to be classified as liquid. A summarized version of the six points presented in the liquid software manifesto is as follows.

- The users should be able to roam between all the computing devices that they have.
- Switching between devices should be casual and hassle-free as possible.
- User applications and data should be synchronized among all the devices.
- Application state should be transferred among devices whenever applicable.
- Switching between devices should not be vendor specific.
- The user should be in control of the liquidity of the application and its state.

These simple set of points not only help us to classify liquid software but also provide a guideline to develop applications which could be liquid in nature. Even though the term "Liquid Software" sounds like just another fancy word that could be used to name applications which support synchronization among devices, this manifesto clearly shows that most of such applications that are available today are not fully liquid. Hence the liquid software manifesto also helps to take away any misconceptions.

Fluid computing [17] symbolizes the replication and synchronization of application state among various devices. The paper mentions that the application state flows from one device to another just like a ‘fluid’. This is indeed another way of referring to Liquid Software. The three main areas that the authors discuss can be briefed as follows.

1. Multi-device applications where few devices could be coupled to behave as a single device
2. Reduction of the effects of unreliable connectivity on ubiquitous devices.
3. Multi-user applications which allow several users to collaborate together on a shared document.

The technical view of fluid computing includes middleware that handles replication and synchronization of the application state among devices. Hence each device carries a copy of application state that allows them to operate autonomously.

Liquid Software, when taken from a user interface point of view goes back to Computer Supported Collaborative Work (CSCW), which focus on enabling collaboration between multiple users [18] in contrast to multi-device usage by a single user. A multi-device, thin-client groupware system which is collaborative and component based is presented in [19]. The main highlight of this work in contrast to liquid software is the fact that it’s simply groupware which has limited support for synchronization of the application state.

Today, Apple has been able to come closest to the liquid nature of applications within its own ecosystem. The Handoff [10] capability among Apple devices allows the users to resume work, which they were doing in one of their devices, in another. For an example, a user can view an image in his/her iPhone and to take a better view on it he/she can switch to his MacBook and still continue to see the same picture. This feature is enabled by tracking the devices by the same apple Id and Bluetooth is used as the communication mechanism between the devices. Furthermore, Apple also supports the universal clipboard where a user can copy from one device and paste the

content in another device in the same ecosystem. If we judge at this level, this is a pure liquid environment, but since it is constrained within the vendor specific environment this violates the Liquid Software Manifesto.

Android's alternative to Apple's Handoff comes in the form of Android Baton from Nextbit[11]. Baton's cloud backend ensures the synchronization of files among registered devices. It can also maintain the state of the application among devices. The fact that Android applications must use this proprietary API to develop applications to exploit these functionalities leaves a major limitation on this approach as well.

The Windows alternative comes in a different shape. Windows Continuum [12] is a small box-shaped device, which can be connected to mobile devices running Windows 10. The mobile devices can connect to Continuum and different device 3 extensions such as screens, keyboards and mouses can be connected to it. This allows switching from a mobile perspective to a desktop perspective as required by the user. There's no need for a backend cloud service to synchronize the data since we are accessing the mobile device through Continuum. This depicts the problem of having to configure the environment, which deviates from the Liquid Software Manifesto.

2.1 Important Architectural Considerations

Liquid Software is not just a new technology, it's a lot more. The term Liquid Software is about constructing a mindset for developing applications which could be executed in multiple heterogeneous devices. It includes making proper architectural decisions which lead to achieving the common goals of a liquid application. An ideal liquid application that makes correct architectural decisions will satisfy the SAFE qualities [5]; Scalable, Adaptive to different environments, Flexible to heterogeneity and Elastic.

There are few interesting research articles on Liquid Software architecture and architectural approaches that can be followed to achieve the fluid nature of software

[2-6]. Following the above identified differentiations, a few essential architectural considerations have been presented in their journal article by Gallibino, Mikkonen, Systa and Taivalsaari [4]. Out of various possible architectural concerns these few will be most essential in developing liquid applications.

- User Interface adaptation

  This will be most applicable when coming to various wearable devices. Even though most of the novel applications support responsive design, it doesn't address the limitations that are there in a different family of devices. Hence it's important to focus on UI adaptation as it is the presentation layer of any application.

- Data and State Synchronization

  It's essential to distinguish between persistent application data and dynamic application state. Persistent data is the static data that is usually saved across user sessions. This is usually available in general applications. But what's hard to achieve is the dynamic state synchronization which means maintaining the runtime information of the application. An example would be to capture the font size and indentation that is used in Google docs of a laptop user to be used in his smartphone to edit the same document in Google Docs. Such limitations do exist in current applications and need to be dealt with. The researchers mention that in order to experience a seamless operation of applications across multiple devices it is required to Identify, Persist, Migrate, Replicate and Synchronize application state across devices.

- Client-Server Partitioning

  It's important for liquid applications to identify and make use of device resources efficiently. The resources of multiple devices differ in a considerable manner. Hence it's important to figure out how the partitioning of the applications can be done. This can also be called as Layering of the application. There can be very thick clients as well as very thin clients. Today, applications that are designed for a family of devices consider

themselves to use options such as Backend as a Service (BaaS) in order to limit the load on the client side.

- Security

  It's important that users are in full control over the transfer of state and data among devices. This should also not become a hassle for the user. Even though downplaying security aspects are commonly seen in liquid like applications, it's important to figure out the means of applying existing security solutions as much as possible in an intuitive manner. There's still a lot of space for research based on the security aspect of liquid software. Other concerns apart, the authentication techniques that can be followed in developing liquid software applications has a huge influence on the overall security and usability. It is common to see many cloud native applications being developed which mimic liquidity for some extent. There are few authentication techniques that could be used for cloud environments. Authentication and Authorization was provided as services in [29] using the service oriented architectural approach. Using these offered services cloud based applications could offer authentication and authorization functionality in their applications. Authentication as a Service is a novel service model which provides many out of the box features in terms of authentication and authorization for application developers. Hence it can be considered as a good candidate to consider when looking for authentication options.

2.3 Liquid Software Design Space

There are various ways of implementing liquid experience in applications. In their journal article by Gallibino, Mikkonen, Systa and Taivalsaari [4], they present a Design space for Liquid Software which shown in Figure 1. A clearer image will be attached to the Appendix for further reference. It shows a top to bottom navigation considering various design alternatives that can be selected in developing liquid software applications.

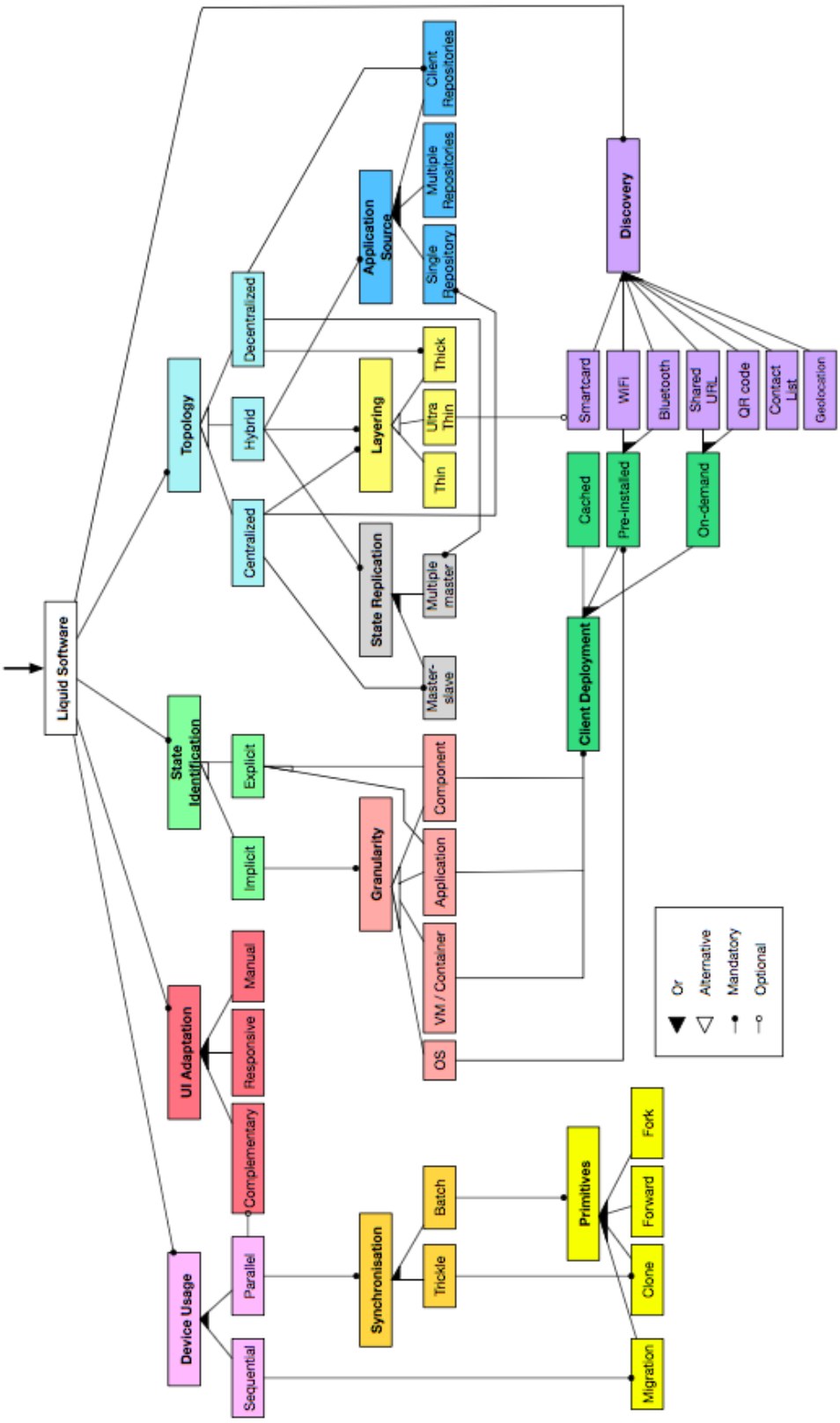


Figure 1- Liquid Software Design Space

The design space diagram is explained further in [3] considering each and every major selection stages. A few of the design decisions which are discussed in this literature are stated below.

- Topology
    - Centralized
    - Decentralized
    - Hybrid

  The easiest to implement a multi-device ecosystem is the centralized approach with a centralized server where clients will rely on screen casting similar to SunRay platform [9]. But such an approach will not be ideal for resource-poor devices, which cannot guarantee a reliable network connection. Hence the Hybrid approach will be a better option, and as shown in Table 1, many technologies have adapted a hybrid approach in this regard.

- Maintaining State of the Application
    - Master-Slave
    - Multi-master

  These two design options focus on maintaining the application state and the data of the system synchronized. Figure 2 illustrates the master-slave and multi-master design options. In a master-slave approach, state replication is centralized. A master database owns the master copy of the state or data and it updates its slave databases instantly each time it updates. In this way, all slave databases and the master database has the same copy of the data or the state at any given time. This approach redeems the master database from a heavy load since the load is distributed among slave databases. This also avoids the Single Point of Failure since data is replicated in both the master and slave databases and a slave database can become a master database in case of a failure in the master database.

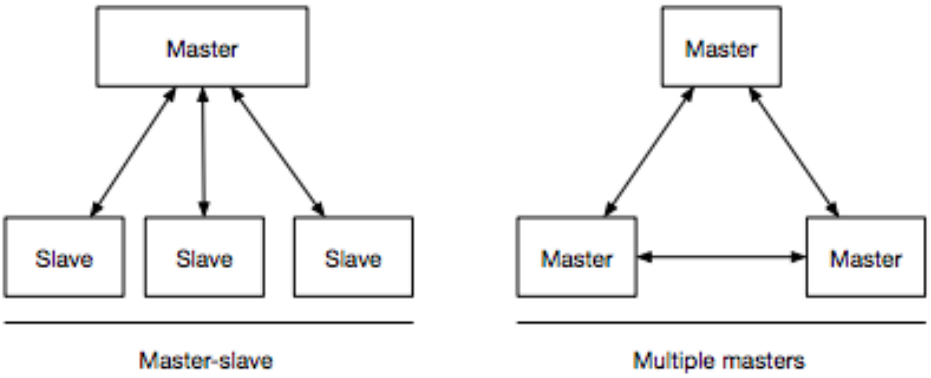


Figure 2- Master-Slave vs. Multi-Master

In the multi-master approach, the state or the data replication is decentralized. There are multiple masters instead of slave databases. Each having the ability to accept or discard state or data changes. But the complexity of this design is high since we have to maintain a consistent state throughout all master databases.

- Granularity

  As shown in figure 1 software applications can be liquid at different granularity levels. The researchers in [4] present a set of layers of software which they propose can be recognized as levels of granularity. Figure 3 presents a couple of granularity alternatives and the layers of the software stack which can be used for migration and synchronization.

  1. Operating system level
  2. Virtual Machine/Container level
  3. Application level
  4. Component level

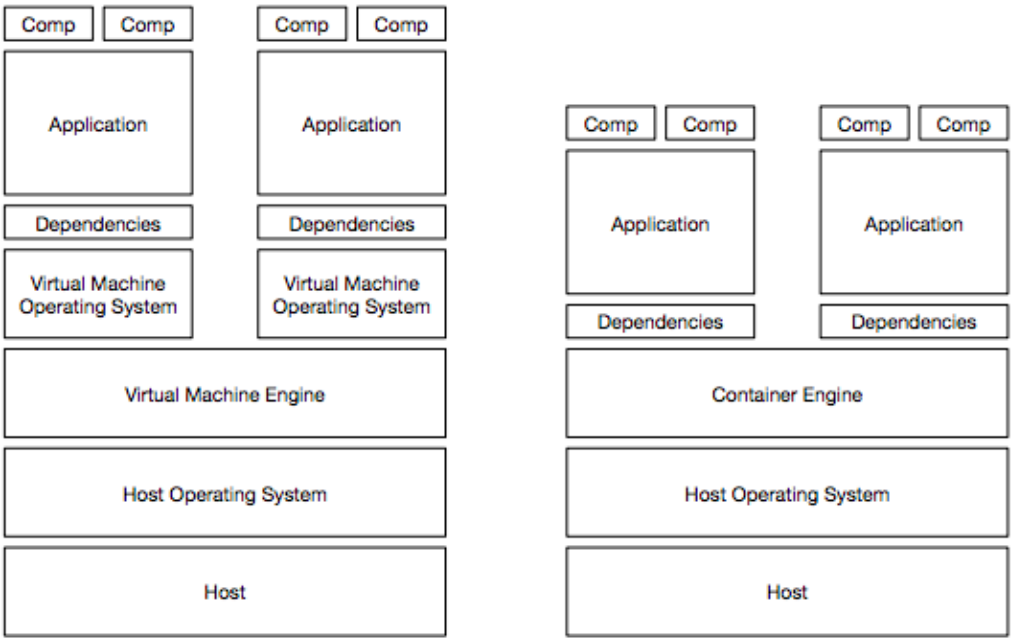


Figure 3- Granularity Alternatives

- Device Discovery

Device Discovery is a mandatory component when it comes to the liquidity of applications. As Discussed in [4] in the modern devices, there exist device side technologies readily available for device discovery purposes. Technologies such as Bluetooth and Wi-Fi are readily available in most modern devices and can be easily used to identify target devices. But when it comes to liquid software applications this adds a huge dependency to the devices and the hardware that the applications operate on. Hence opting for options such as QR codes and shared urls would mean that the applications are independent of such device hardware. In the research article on Architecting Liquid Software [4], three main Discovery concepts are discussed.

1. Existence Discovery

   The minimum requirement for fulfilling liquidity in applications is identifying all available devices. As discussed previously technologies like Wi-Fi and Bluetooth can be used for this purpose by setting up a Local area network. This doesn't require the access to the internet and

devices are identified by their MAC Addresses. And when access to the internet is available options such as QR Code and Shared URLs could be used by allowing communication through the internet. This approach will use IP addresses in order to identify the devices.

2. Location Discovery

   Even though maintaining relative location of devices is not a mandatory requirement in providing liquidity to applications, it could seriously improve the liquid experience in a ubiquitous environment. Features offered through Apple Handoff [10] leverage such functionality for sharing content in between Apple devices by using hand gestures. Modern devices come with GPS capabilities which can be directly used to come up with solutions with this regard. But since offering geo location services affect the power consumption of devices, it should be properly balanced. Liquid Software applications that are based on web technologies can leverage high energy consuming geo location services [4].

3. Ownership Discovery

   Mapping between users and their devices needs to be handled carefully and transparently when it comes to liquid Software. As mentioned in the liquid software manifesto [1], application users needs to be in full control of the device transition process. Furthermore, it is important to identify and keep track of devices owned by a particular user when it comes to scenarios such as simultaneous device usage [4]. In ownership discovery it is expected that users are properly authenticated into each device in order to ensure application security.

Table 1 gives a summary of the liquid software design space and the positioning of technologies and proof of concept applications in it.

Table 1- Liquid Software Design space and positioning of the technologies

		Sun Ray [41]	Joust [4]	Fluid Computing [13]	Apple Continuity [21]	Android Beam [22]	Cloudberry [18] Cloudbrowser [44]	Continuum [23]	XD-MVC [26]	Liquid.js DOM [32]	Liquid.js for Polymer [33]
		1997	1999	2005	2014	2014	2014	2015	2015	2016	2016
Architecture	**Topology**										
	Centralized	✓			✓	✓	✓				
	Decentralized			✓							
	Hybrid		✓					✓	✓	✓	✓
	Application Source Topology										
	Single repository	✓	✓		✓	✓	✓		✓		
	Multiple repositories							✓		✓	✓
	Client repositories			✓							✓
	State Replication Topology										
	Master-slave	✓			✓	✓	✓				
	Multiple masters		✓	✓				✓	✓	✓	✓
	Layering										
	Ultra Thin Client	✓					-	✓			
	Thin Client						✓				
	Thick Client		✓	✓	✓	✓			✓	✓	✓
	Client Deployment										
	Preinstalled	✓	✓	✓	✓			✓			
	On-Demand					✓			✓	✓	✓
	Cached						✓			✓	✓
	Granularity										
	OS	✓						✓			
	VM/Container										
	Application		✓	✓	✓	✓	✓			✓	
	Component						✓		✓		✓
State	**State Identification**										
	Implicit	✓						✓			
	Explicit		✓	✓	✓	✓	✓		✓	✓	✓
	Synchronization										
	Trickle	✓	✓	✓			✓	✓	✓	✓	✓
	Batch			✓	✓	✓	✓			✓	
Liquid User Experience	**Device Usage**										
	Sequential	✓		✓	✓	✓	✓		✓	✓	✓
	Parallel		✓	✓				✓	✓	✓	✓
	UI Adaptation										
	Manual		✓	✓					✓	✓	✓
	Responsive	✓					✓		✓	✓	✓
	Complementary				✓	✓	✓	✓	✓		
	Primitives										
	Forwarding	✓	✓					✓	✓		
	Migration				✓	✓	✓			✓	✓
	Forking								✓	✓	✓
	Cloning			✓			✓		✓	✓	✓
	Discovery										
	Shared URL		✓				✓	✓	✓	✓	✓
	QR Code								✓	✓	✓
	Bluetooth			✓	✓	✓					
	WiFi			✓		✓					
	SmartCard	✓									
	Contact List								✓		
	Geolocation								✓		

2.4 Architectural Examples and Proof of Concepts

There are few Architectural examples and proof of concepts discussed in [1]. Even though the name “Liquid Software” itself sounds novel, there have been several attempts to achieve liquidity in applications since the later part of the 20th century [7, 8, and 9].

- Sun Ray platform [9] by Oracle brings out a semi-liquid approach where users are able to transfer their entire network computing sessions from one terminal to another. This is achieved by “screen-casting” where compressed screen images from the servers are transmitted to the Sun Ray terminals live. Since no data are transferred away from servers, it is identified as a very secure approach, but the main downside of this approach is that it requires a reliable network connection. This dependency leads this attempt to the category of semi-liquid computing.

- Cloudberry [13], which was developed by Nokia in 2009- 2011, is a proof of concept HTML5 mobile device platform where the device’s user functionality is cached dynamically from the Web. It has the capability to download and cache all the applications including the entire top-level user interface. Master-Slave database architecture is used with push notifications to dynamically synchronize the application state.

- A pee-peer multi-device ecosystem [14] was created by using the world-and-wormholes metaphors of the Lively Kernel [15]. The Lively Kernel implementation was extended to be able to distribute in multiple computers. The world is a visual container that hosts several lively kernel applications and objects concurrently in live. Wormholes metaphors act as a bridge to transfer such applications and objects between these diversified and different worlds [20, 21]. Kernel implementation was previously kept within the bounds of worlds that were hosted in the same computer, yet elongated at a later stage so that wormholes can be related with Lively Kernel worlds

situated in distinct computers. At this implementation level, a shared server was utilized instead of utilizing P2P access (http://www.w3.org/Security/wiki/Same_Origin_Policy). In the current era, it is possible to implement a peer-to-peer multi-device environment with the help of a WebRTC (https://webrtc.org/) which is an open and free real-time communication through simple APIs.

- A liquid Software framework was developed using HTML5 mobile agents [16] where web applications can store their internal state in a server for the future reference. Hence their executable code can be shifted to a different computer along with the saved state.

- Liquid.js for Polymer [6] is a framework that enables development of liquid web applications using web components. Liquid.js operates at the component level granularity. It relies on Yjs [28] to handle the state synchronization. The three main use cases of this framework are sequential screening, simultaneous screening and collaboration. Liquid components defined in the framework are pieces of mobile code which is a combination of JavaScript and UI components which can render on the web browser when required using HTML imports. These liquid components are built on top of the Web components using the Polymer syntax.

2.5 Authentication as a Service

Authentication as a Service (AaaS) is generally based on centralizing the authentication logic and presenting it in terms of a service to be consumed by other applications. Generally a robust AaaS should support authentication and authorization protocols such as OpenID Connect [30], OAuth 2.0 etc. and would typically provide the identity services through a well-defined API. There are two options for enabling AaaS in a software ecosystem. One is to come up with an Authentication server from the scratch which is compliant to the standard

authentication protocols and implement the ecosystem around it. While this approach allows complete flexibility, it might not be the ideal option when there are ample off-the-shelf authentication server implementations available unless we are looking at a completely customized requirement. Keycloak [23] is one of these off-the-shelf implementations that are available. It is an open source identity and access management solution. It provides support for standard protocols such as OpenID Connect, OAuth 2.0, and SAML. It is capable of providing not only authentication but also role based authorization as a service. It also has modern features such as Single Sign-On and Identity brokering that are expected by a novel authentication platform. Using Keycloak, applications can easily be protected by a simple configuration.

CHAPTER 3

METHODOLOGY

The main focus of this research is to analyze architectural design decisions which are applicable for the application-level granularity of liquid software. It is one of the main attempts of this research to work on the security aspect of liquid applications which is still an immature research area. In contrast to the liquid software design space presented in [4], this research tries to improve on top of that by including high-level architectural decisions which target application-level granularity. For an example, various service models such as Backend as a Service (BAAS), Software as a Service (SAAS) and Platform as a Service can provide out of the box features for software development. Authentication as a service (AaaS) is also another service model which provides authentication out of the box for application development. Therefore we propose an Authentication and Device Discovery as a Service model which combines the core functionalities of authentication and device discovery. One of the main reasons behind this combination is to provide as much liquidity as possible for application developers to be used out of the box. As mentioned in the introduction section, the combination of authentication and device discovery would mean the following points of the liquid software manifesto[1] is covered.

- Effortless roaming is expected between the user-owned devices.
- The roaming process should be casual and hassle-free as possible.
- Roaming between devices should not be affected by vendor-specific limitations.

These are three of the six statements given in the liquid software manifesto which helps identify software applications that are liquid in nature. The proposed combination of authentication and device discovery as a service not only ensures the coverage of the above 3 statements but also partially covers the part where the users are in full control of the liquidity and the state of the application. Hence the applications that consume the authentication and device discovery as a service automatically embrace partial liquidity.

Some authentication services could only be offering authentication while ignoring user management altogether. But having the variety to choose between them would

be the ideal scenario. Some teams would want to maintain their own user management store while only consuming the authentication services through an authentication server while some teams would want to kick start the entire user management and authentication process by consuming such functionality from service providers. Therefore having a mix of both possibilities is always essential to a service model as it broadens the target audience.

Microservice architecture is commonly used in modern application development where functionality can be modularized. Some obvious advantages of microservices are independent development, deployment and maintenance. Using microservices architecture independent reusable components could be developed with less coupling and more cohesion. But authenticating requests to these services in a liquid environment should be done carefully. Furthermore, end users shouldn't be hassled by requesting them to authenticate themselves into each and every service they consume. It is the expected behavior that users should be able to consume all services provided by an application with a single sign-on, irrespective of the backend architecture. This should be true even if the user decides to switch between devices. By following the liquid software design space [3] one can come up with a set of possible architectural routes, depending on the application requirements. This way we can get a high-level view of the liquidity of the application in the early design stages. This not only adds value to the whole software engineering process but also makes the way for identifying major components or technical aspects that can make huge impacts on the system in the early stages. One of the least discussed topics when it comes to architectural design decisions is the security aspect of liquid software applications. As mentioned before, it's most common to identify security risks when making a system flexible to adhere to liquid concepts. The proposed solution not only provides user registration and authentication out of the box but also provides service level authentication. Using this authentication component, applications that follow microservices architecture can also protect all their separate services. This guarantees that individually functional microservices are well secured. And also additional effort is not required to authenticate requests that are coming in towards these individually deployed services.

3.1 Basic Design Decisions

One of the most important aspects of a solution is to make proper design decisions as early as possible. Proper design decisions reduce the risk of disaster and reduce the development time by a considerable amount. The exact opposite applies to bad design decisions or vague decisions. After a thorough study of the design space of liquid software [4], few areas were identified under the application level granularity, which could be combined to provide a liquid like experience to an end user. This research basically targets the application level granularity of the liquid software design space. Other granularities such as operating system level, container level are out of the research scope and will not be discussed in detail. The main objective of this research is to assist the application developers in the authentication and device discovery process of liquid software applications. The proposed Authentication and Device Discovery as a Service is consumable by any web application.

It's really important that we identify and break down a solution to small components such that it becomes easier to implement these components in a broader scope independently. There are few other aspects of liquid software architecture which could be decided and handled by the application developers themselves. The layering of the application is one of those design decisions that the application developers would want to make for themselves. It could depend on the target devices of the application. If the target devices are resource poor the better option would be to opt for a thin client where the majority of the business logic would be implemented on the server side while only the presentation would be done on the client side. But with the recent developments in mobile technology many mobile devices now come with a lot of computing power. Hence it is most common to see many thick client applications in the modern market. There are always certain decisions that developers would want to make based on particular requirements, but our main goal is to gather a set of core components that could provide liquid properties for an application.

3.2 Software Development Process

As in almost all cases, following a suitable software development process is essential for a successful project. Since this research work is based on the development of a service model to support liquid application development, this project would also benefit a lot from a proper Software Process. Considering the breadth of the scope and the modularity of features, the iterative model of development looked to be the ideal candidate for the Software Development Life Cycle of this project. Following the iterative model, the following services will be implemented.

1. Authentication Service
2. Device Discovery Service
3. API Gateway
4. Service Registry
5. Liquid QR Code Scanner
6. Proof of Concept

Since these are individually independent components and has very less coupling, these could also be implemented in parallel if required. How the software development life cycle of this entire project spanned out is shown in Figure 4. As illustrated in Figure 4, after the completion of the Authentication service, then the Device Discovery service is started as the second iteration. As the 3rd iteration, an API Gateway and a Service Registry is implemented. As the 4th and the last iteration, the Liquid QR code Scanner and the Proof of concept application which consumes the Authentication and Device Discovery as a service is implemented. The functionalities of the authentication service are required to authenticate requests received by both Device Discovery and the API Gateway. Due to this fact and since the authentication service is one of the major outcomes of this research, its design and implementation were carried out as the first iteration. By following the incremental model of development, each iteration will produce a well-designed, implemented and tested solution. This will make sure that defects and flaws are identified as early as possible in each component as they are being developed.

Furthermore, integration tests can be carried out incrementally in each iteration making sure that all integrations are working as expected. The incremental model supports early identification of integration issues and hence guarantees a working solution at the end of each iteration.

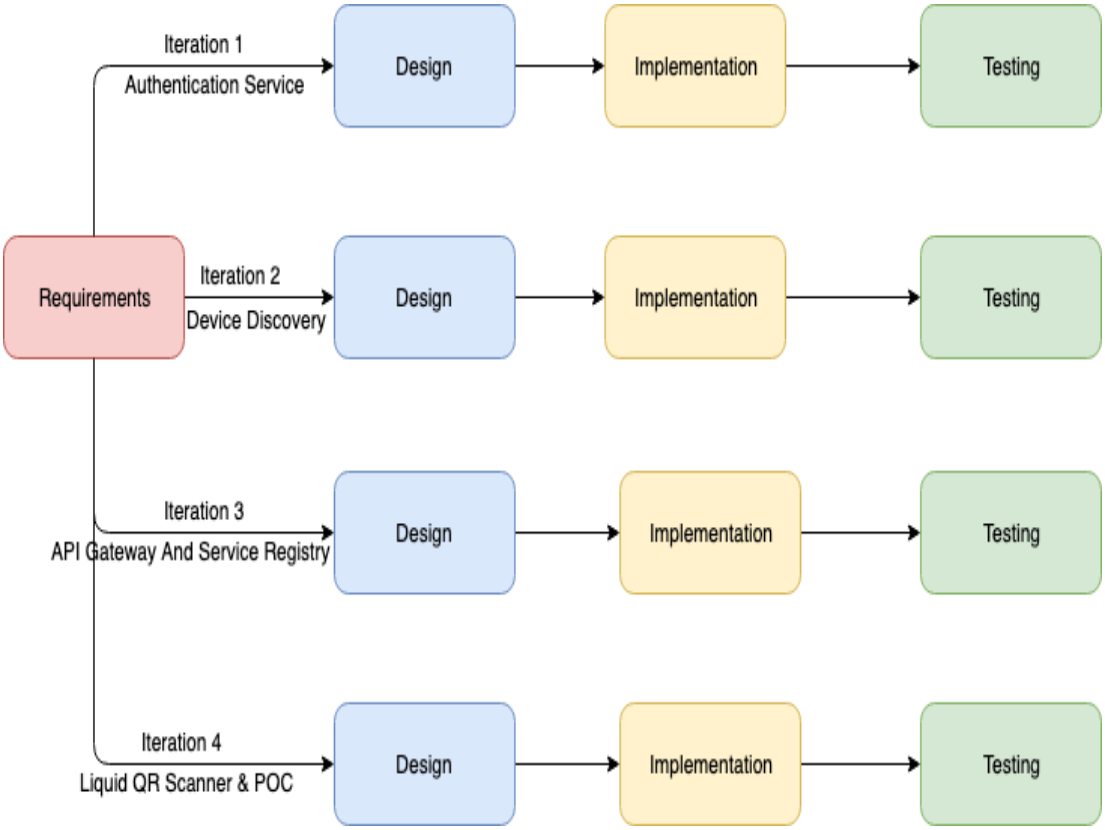


Figure 4- Software Development Life Cycle

3.3 User Authentication

One of the least spoken, but one of the most important aspects of liquid software is security. Much research work is required on this domain since it could be easily downplayed when trying to achieve liquidity. Security and Privacy are largely spoken topics in the modern world. When it comes to liquid software it's essential that proper user authentication and authorization is performed as breaching one device could compromise all the devices of a user. But authentication should ideally

be hassle-free as possible. According to the liquid software manifesto [1], switching between user devices should be a casual process.

Even though authentication of users is a primary concern in any application which has such a requirement, so are the authentication of web requests received by the application servers. Authenticating a user to an application is a transparent process for the end user while authentication of web requests generated by their actions is not. But once a user login to a system, it is the expected behaviour that all his actions are properly authenticated. This fact should not change based on the application architecture, or else the user experience would definitely get affected. The authentication component of the service model that is presented in this work, would handle the following aspects of authentication.

1. User Registration
2. User account management
3. User Login with Single Sign-On
4. User Authorization
5. User Federation
6. Service Level Security
7. Identity brokering and Social Login

It is most common and effective to use already tested functionality over reinventing the wheel. If the requirement is not at all generic, the option to write your own functionality is justified. But when it comes to Identity and Access Management, there are ample solutions available. The feature-packed authentication service is built on top of Keycloak which is an open source Identity and Access Management solution. The solution is more efficient when built on top of a stable, well-tested solution rather than reinventing the wheel. It will be further discussed in the implementation section on how Keycloak plays a major role in this solution. The capability to work with different Identity and Access Management solutions is a possible future improvement for this solution.

3.4 Device Discovery

One of the main important aspects of liquidity is the ability to discover devices. Ubiquitous computing is all about providing the power of computing whenever wherever. But in order to provide such an experience, applications should be operable from different devices. The most initial step to providing a liquid experience to users is device discovery. How liquid software becomes aware of possible devices that it can run on is one of the most important aspects that need to be considered. A proper discovery mechanism needs to be based on factors such as the proximity of the device, reachability of the device and availability (online/offline) of the device [4]. When it comes to application level granularity, it is really complex to maintain the availability of a device unless that particular application is running at least in the background of that device. Hence considering several technologies that are readily available on the device is one of the alternative options. Both Bluetooth and Wi-Fi discovery mechanisms are well tested and ideal options if we consider moving to device-side technologies. Another alternative option is to use shared URLs or QR codes which provide maximum control to the application code itself without having any dependency with the device hardware. Ownership discovery is essential to the security of the liquid experience. In the proposed design, when combined with the authentication service, the user will be in full control of the device discovery process as it is the user who decides which device he/she needs to move in to. This is done with minimal configuration using the authentication service. Once a user gets authorized to the application, he/she can use the bearer token offered by the authentication service to roam between his/her devices. This process could be handled by scanning a QR code or using a shared URL. This token will be under an expiration interval which could also be changed as required. Figure 5 shows a sequence flow of the device discovery process of an application which consumes the proposed device discovery service. How to consume these features offered by the service model will be elaborated in the implementation section. As shown in Figure 5, a user who is trying to access the client application which is protected by an authentication server first needs to authenticate himself with the Authentication

server by providing his credentials. Once authenticated the user will be redirected to the application, with a bearer token. Now that this user is authenticated, until his token expires, he is allowed to use the application and its services. When a user needs to register a new device and continue his usage of the application from that device, he would request a QR code from the device discovery service. Once his bearer token is validated with the authentication server he will be granted with a QR code. The provided QR code has the following data embodied inside of it.

1. Redirect Url
2. Authentication session cookie
3. An endpoint to register a new device
4. Metadata

Since this is a customized QR code it needs to be scanned by the Liquid QR Code Scanner. Once scanned it will perform the device registration process through the device discovery service and will redirect to the provided url with the bearer token and the provided metadata. The metadata can also be used to transport the liquid state of the application. It totally depends on the consumer of the service whether to use it to transfer state or some other metadata.

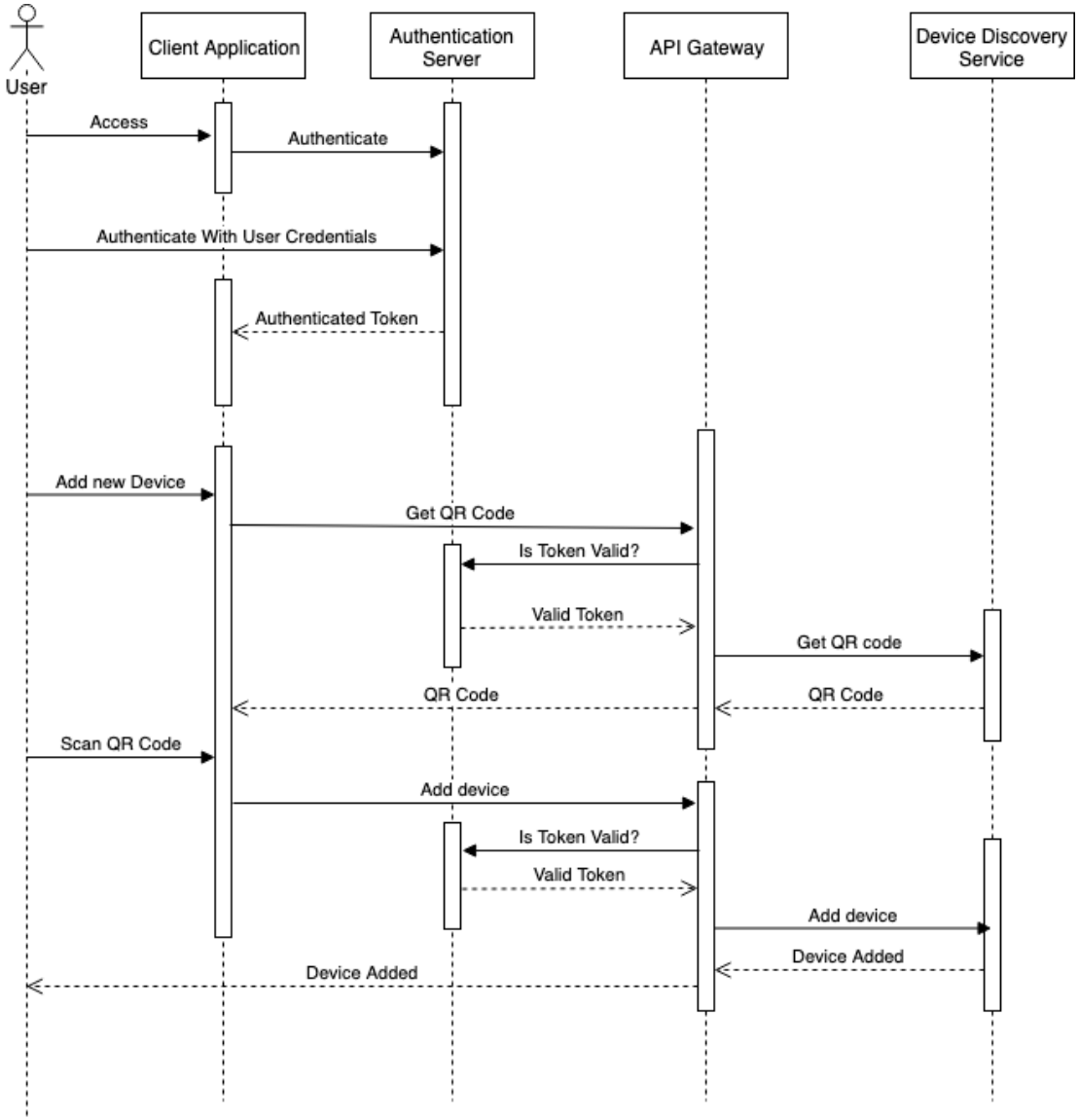


Figure 5- Device Discovery Scenario

3.5 Authentication and Device Discovery as a Service

The main outcome of this research is the service model which provides authentication and device discovery as a service. Even though these two are combinedly presented, both authentication and device discovery are two independent components. The device discovery service adds more value to an application when combined with authentication. In the proposed solution the authentication service is mainly based on Keycloak which is an open source Identity and Access Management solution. The concept behind the proposed authentication model is shown in Figure 6. As shown in Figure 6, every application which consumes the service is identified as a “Realm”. A particular user of a certain application is identified as a “Tenant” in a particular realm. Each realm (application) has its own user store where the credentials of each user (tenant) of that application are stored. Therefore development teams can choose to use this service with multiple applications without any issue. Furthermore, the underlying resources will afterwards be secured with the user tokens that are based on each realm and tenant. That’s what is illustrated as resources being protected inside a realm and a tenant in Figure 6.

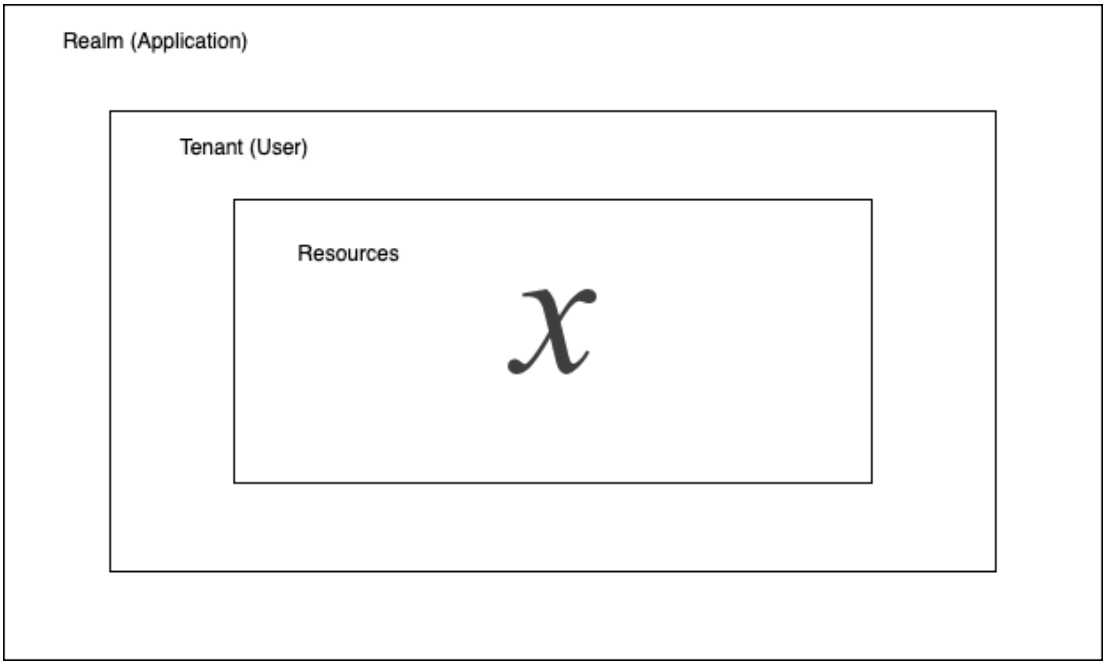


Figure 6- Basic Security Concept

Since the solution is wrapped around Keycloak, all of its features are readily available to the end users. Application developers can consume the features offered by the authentication service in a number of ways depending on their other architectural decisions. Few possible scenarios can be identified as follows.

- If we consider an application architecture that is layered as client-server, application developers can choose to protect the client application by directly connecting to the Keycloak server and using it's login/register page. This is a good starting point if you just need a login page for your application. This option is ideal for thick clients where most of the application logic is in the front. The basic login page provided is highly customizable, therefore development teams could basically adapt its functionality while also maintaining a unique look and feel. Steps required to securing a front end application directly with Keycloak will be later discussed in the implementation section.

- Development teams could also access the authentication service directly through its well defined Rest API. This is also a good option for thick clients.

When used in combination, the authentication service could provide authentication for the front end application and then afterwards guarantee the security of its service requests. All microservices that are consumed by a client application which is protected by the authentication service should also be protected by the same in order to maintain a proper authentication chain. But having to include configurations and boilerplate code in each and every service could be duplication of work. And also it would add on to the maintenance effort. Therefore an API gateway is introduced to intervene each and every request made to the backend services. Figure 7 shows the high-level architecture of the proposed authentication and device discovery service model. As shown in the figure, it is the API gateway that is actually being protected by the Authentication server. When an Http request is received by the API gateway, it first verifies its bearer token with the authentication server. Once the authentication server acknowledges it as an authenticated token, the API gateway looks up the service registry and serves the requested resource accordingly. Considering the consumption scenarios of the services that are offered through the service model, the API gateway is also equipped with a load balancer. This guarantees the service requests are equally distributed among multiple instances. In order to cater the requests, the API gateway refers to a Service Registry. Therefore each service that is being offered needs to be registered with the service registry in order to be identified by the API gateway. As shown in Figure 7, currently we only have the device discovery service offered with the combination of the authentication server. But in the future, more services that are generic to liquid applications could be integrated into this solution. Therefore this is a scalable solution having space to accommodate more generic services.

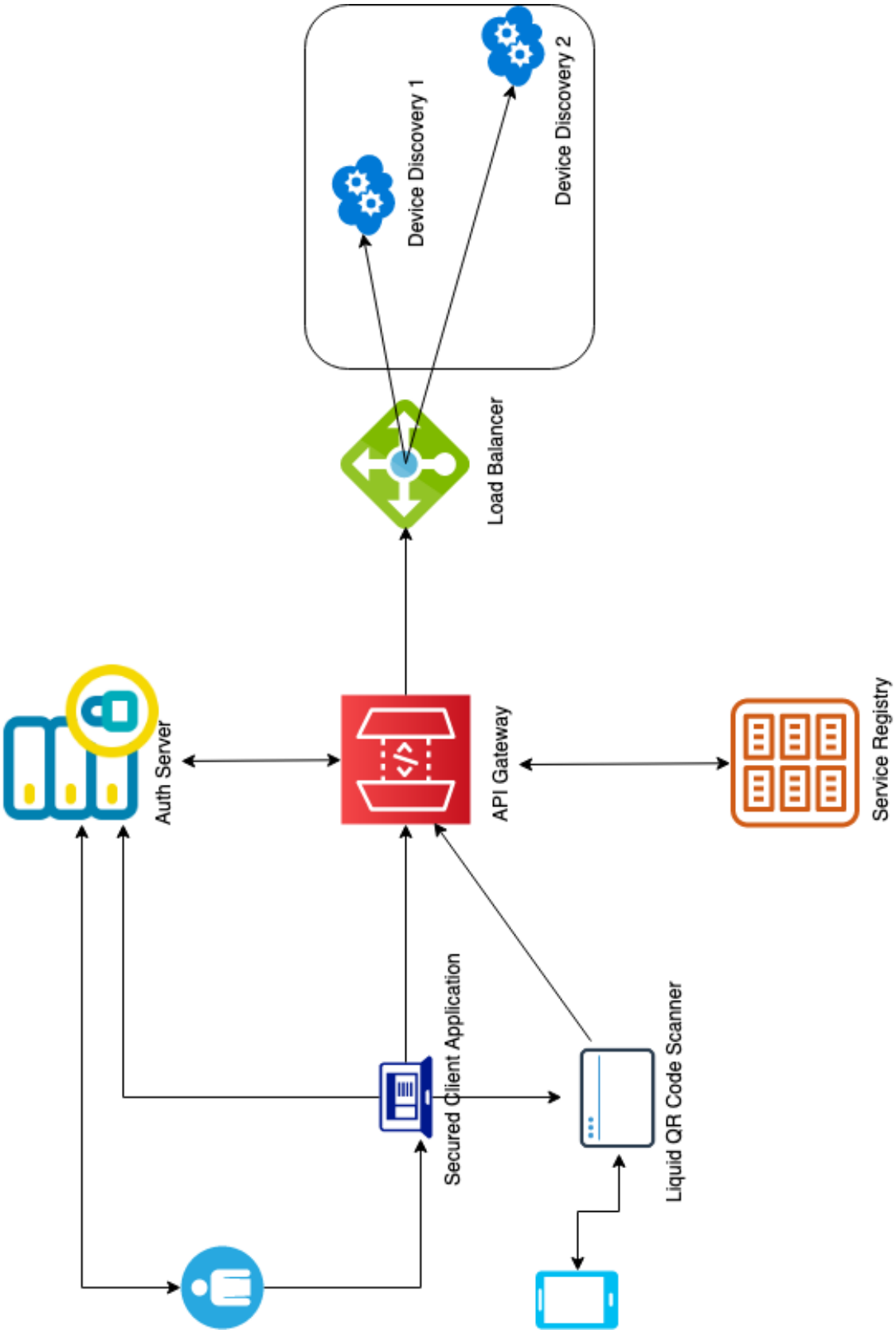


Figure 7- High-level design of the Authentication Service

As previously illustrated in a sequence diagram in Figure 5 the service consumption sequence can be broken down into the following points. The following points cover the basic scenario of adding a new device to an existing user.

1. The user accesses the client application and tries to add a new device.
2. The client application sends in a GET request for a QR code which includes the bearer token and the redirect url to be loaded from the target device.
3. The API gateway intercepts the request since all requests that are received needs to be authenticated.
4. The API gateway validates the bearer token with the authentication server. Once it is verified as authenticated, the API gateway looks up the service registry for the requested resource.
5. Once identified, the API gateway sends in the request and gets the response from the service. This generated QR code is now displayed in the client application.
6. The target device should first load the liquid QR code scanner in order to scan the QR code generated by the source application.
7. The Liquid QR code scanner then sends in a request with the bearer token to register the new device.
8. The API gateway again does the same routine and completes the request. Now once the request is successfully completed, the target device will now be redirected to the provided url.

This is the entire authenticated device discovery process provided by the proposed solution as shown in Figure 7. In the implementation section, we will look at how these services were developed and how they can be integrated into a web application.

CHAPTER 4

IMPLEMENTATION

The Authentication and Device Discovery Service model was implemented mainly based on the Spring Cloud [22] technology stack. Furthermore, as previously mentioned Keycloak [23] is used as the User Authentication and Authorization server. Both the Liquid QR Code scanner and the Proof of Concept client application was developed with React [24] components. The high-level diagram of the technology stack that was used during the implementation of the authentication and device discovery service model is shown in Figure 8. Figure 8 is a technology replacement for Figure 7. The API gateway is based on Netflix Zuul and is a Spring boot application. Netflix Eureka service is used as a service registry. Device Discovery service is also developed as a spring boot microservice.

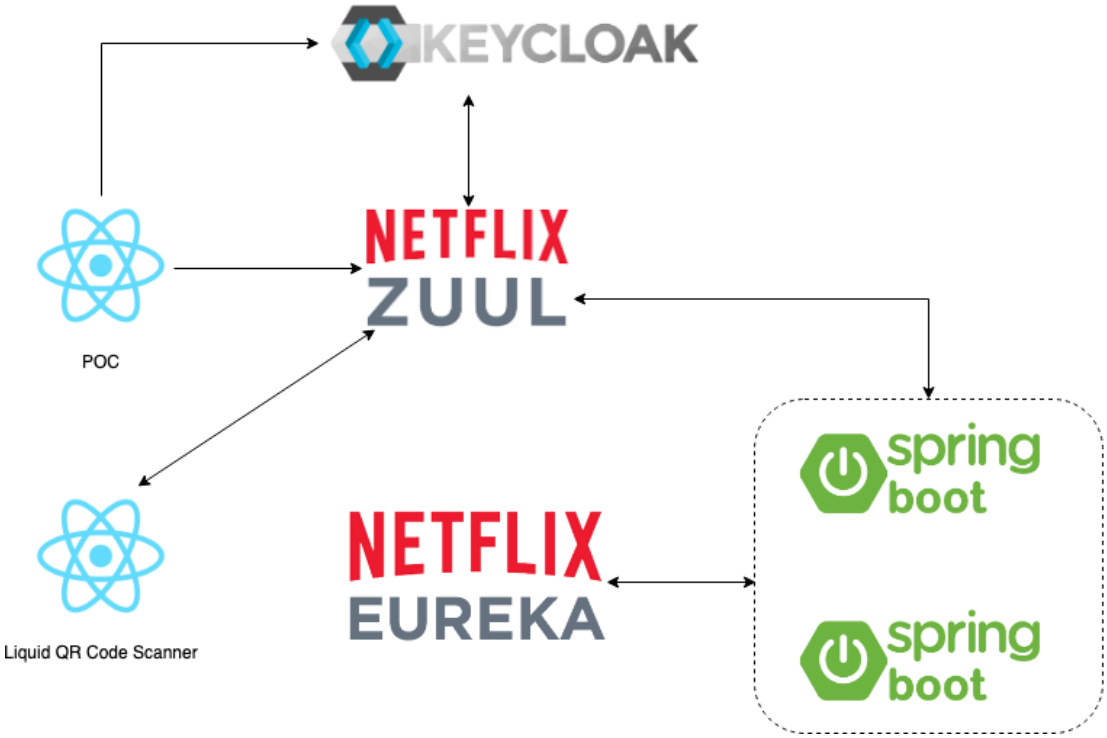


Figure 8 - Technology Stack

4.1 Users and Devices

The user registration process may or may not be handled directly using inbuilt Keycloak functionality depending on the service consumers' preference. There could be applications that require an entire user authentication process out of the box whereas there could also be applications that would like to handle their own user management process while consuming services of an external authentication server like Keycloak to generate authentication tokens. Since our technology stack is mainly based on Spring technologies, we decided to adapt the OAuth 2.0[25] as the security protocol which is supported by both Keycloak and the Spring Framework. The decision of taking full advantage of user management features offered by Keycloak or having an application user management process can be technically described using OAuth 2.0 grant types [26]. According to OAuth 2.0 grant types the two mechanisms can be identified as follows.

- Third Party Applications - Authorization code grant type and implicit grant type
    - When it comes to third-party applications where user's confidentiality needs to be maintained, the Authorization code is the most commonly used grant type. Users are not required to share their credentials with the application which they want to consume resources from. This is a redirection-based flow where the application must be able to interact with the users' web browser.
    - The client application (frontend) makes a request to the User Authentication and Authorization server on behalf of the user.
    - The Authentication server either redirects to an authentication page of itself or a different third party where the user authenticates himself.
    - The Authentication server returns an authorization code with a redirect url if authenticated from a third party.
    - The client application uses the authorization code and an application identifier to request an access token from the Authentication server.

    - The Authentication server verifies the authentication code and returns an access token.
    - The same procedure is applicable for implicit grant type except it doesn't exchange an authorization code to issue an access token.

- First Party Applications - Password Grant Type
    - This is the preferred choice when the users trust the client application. When it's directly the application that is intended to be used by them that they want to authenticate to, they could do so by providing user credentials directly to the application. Applications at this point may be considering maintaining their own user stores.
    - Users provide their credentials to the client application (frontend), commonly done with a user login form.
    - The client application generates a POST request with the credentials to the Authentication server.
    - The Authorization server validates the user and returns a valid JSON Web Token (JWT).

By using either of these grant types, applications may access the protected backend services by using the issued access tokens.

For an application that consumes the device discovery service, the following functionalities were identified as fundamental features that need to be implemented.

- Register a new user with a device
- Register a new device for an existing user
- Modify an existing device
- Unregister an existing device
- Get a QR code for device registration

In order to cater to the above-mentioned functionalities, the following endpoints were implemented. Shown in Table 2 are the endpoints for each functionality, its request type, request parameters and response.

Table 2 - Service Endpoints

Functionality	Rest Endpoint	Request Type	Request Parameters	Response
Register a new user with a device	/discovery/user	POST	username: String device: Device	userId: String deviceId: String
Register a new device for an existing user	/discover/device	POST	userId: String device: Device	device: Device
Modify an existing device	/discover/device	PUT	userId: String device: Device	device: Device
Unregister an existing device	/discovery/device/unregister/{deviceId}	POST	userId: String deviceId: String	device: Device
Get Device Register QR code	/discovery/device/register	GET		qrCode: String

Out of these endpoints, the most important one is the GET request for a QR code. It is to be used by the front end of the application to register a new device for an existing user. In fact, all the other endpoints depend on this particular QR code. Googles' Zxing library was used to generate QR codes in the proposed solution. Zxing is an open source, multi-format 1D/2D barcode image processing library implemented in Java. The following maven dependencies need to be added in order to include the zxing functionality in an application.

```
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>core</artifactId>
  <version>3.3.0</version>
</dependency>
<dependency>
  <groupId>com.google.zxing</groupId>
  <artifactId>javase</artifactId>
  <version>3.3.0</version>
</dependency>
```

The following method generates a byte stream of a QR code generated using the functionalities offered by the zxing library.

```
private byte[] getQRCodeImage(JSONObject source, int width, int height) throws
WriterException, IOException {

   QRCodeWriter qrCodeWriter = new QRCodeWriter();
   BitMatrix bitMatrix = qrCodeWriter.encode(source.toString(),
BarcodeFormat.QR_CODE, width, height);
   ByteArrayOutputStream pngOutputStream = new ByteArrayOutputStream();
   MatrixToImageWriter.writeToStream(bitMatrix, "PNG", pngOutputStream);
   byte[] pngData = pngOutputStream.toByteArray();
   return pngData;

}
```

Using the above method a QR code can be generating by passing the required data in the form of a JSONObject. The JSONObject would include the following.

1. Redirect URL
2. Authentication session cookie
3. The endpoint to register a new device
4. Metadata

The front end consumers may use the QR code as preferred in order to offer their users the ability to register new devices.

4.2 Securing the Application Front end

Given that there's already a "Realm" created in Keycloak in the name of the application, the first step that is required in order to secure the front end of the application is to register it as a client in the authentication server. In our design, we have used Keycloak as our authentication server. Hence a client should first be registered in the keycloak server. This can be done either by accessing the admin console of the keycloak server or by calling the particular endpoint defined in the authentication and device discovery service. Figure 9 shows the settings page of an added client in the admin console where the security of a front-end of the application could be configured.

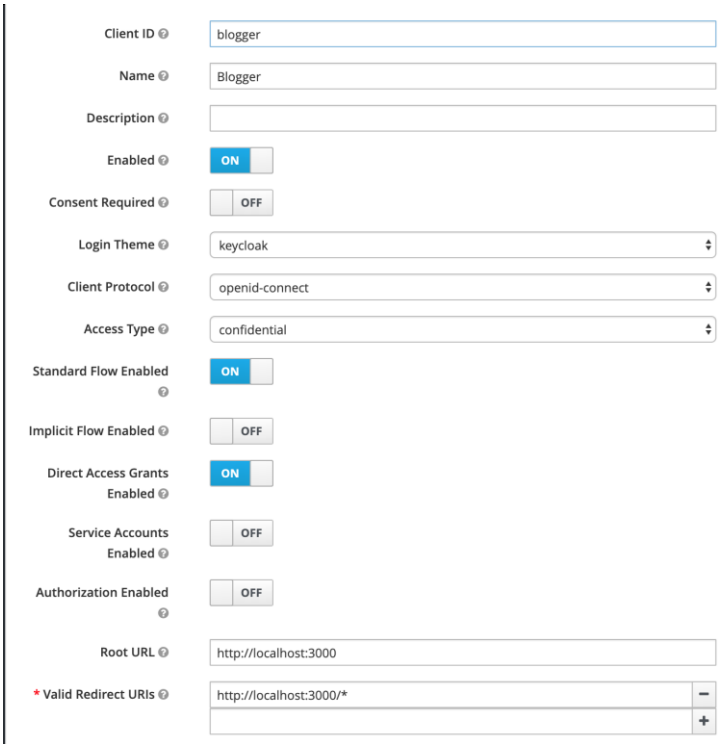


Figure 9- Client Settings

When a client is successfully created in keycloak, the keycloak.json file shown in Figure 10 needs to be included in the public folder of the source code. The contents of the keycloak.json file are almost self-explanatory. The realm stands for the application name, the auth-server-url stands for the url of the authorization server and

the resource for the name of the front end of the application which was added as a client in keycloak. The secret in the credentials section is based on the access type selected when creating the client.

```
{
  "realm": "LiquidPOC",
  "auth-server-url": "https://localhost:8080/auth",
  "ssl-required": "external",
  "resource": "liquid-poc",
  "credentials": {
    "secret": "f8a22e21-c538-449b-bb5c-82865c1fe23c"
  },
  "confidential-port": 0
}
```

Figure 10 - Keycloak.json Configuration

With this simple configuration, now the front end of the application is successfully secured with keycloak authentication. Now a user who tries to access the application will be first redirected to keycloak following the authorization code grant type. A user will have to provide user credentials to keycloak and authenticate himself first in order to be redirected to the application front end as an authenticated user.

4.3 API Gateway

As illustrated in the high-level design diagram in Figure 7, each and every request made to the authentication and device discovery service must pass through an API gateway. This design not only enables us to add request filters at the earliest stage possible but also limits the boilerplate security config code which would have to be included in each and every microservice. In this design, the API gateway is also secured with keycloak. In order to use keycloak functionality in a spring boot application, the following maven dependency needs to be added.

```
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-spring-boot-starter</artifactId>
  <version>${keycloak.version}</version>
</dependency>
```

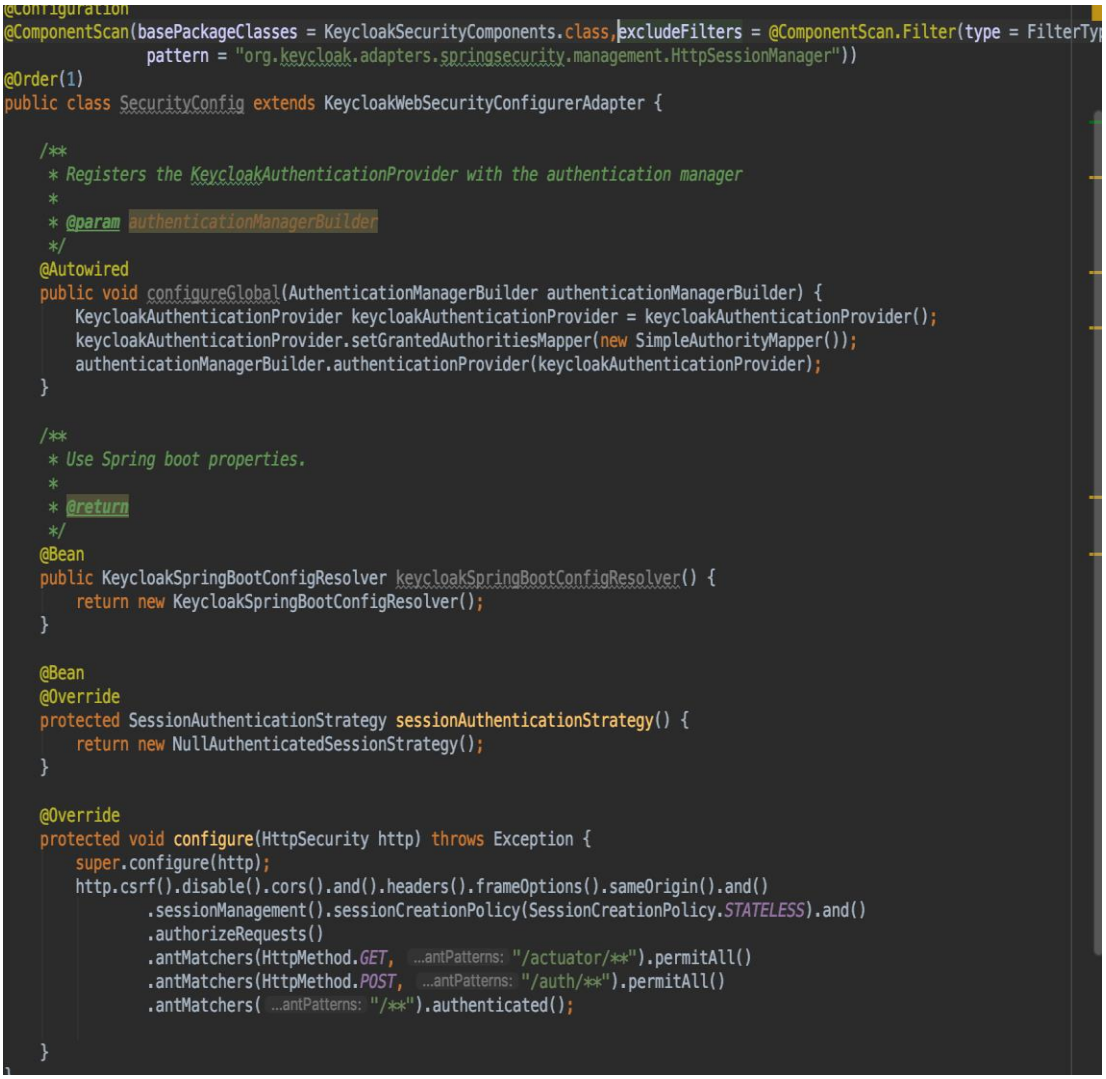
The API gateway implemented for this solution is based on Netflix Zuul. Zuul is an edge service that has the capability to provide dynamic routing, security, monitoring, resiliency and more. In the API gateway implementation, we have used Zuul for the sole purpose of securing all the backend services and to act as a reverse proxy. The complete list of dependencies and the project structure of the API gateway is shown below in the pom.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>2.1.4.RELEASE</version>
     <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.liquid</groupId>
  <artifactId>gatekeeper</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>gatekeeper</name>
  <description>Demo project for Spring Boot</description>

  <properties>
     <java.version>1.8</java.version>
     <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
     <keycloak.version>4.8.0.Final</keycloak.version>
  </properties>

  <dependencies>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
     </dependency>
     <dependency>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.keycloak</groupId>
        <artifactId>keycloak-spring-boot-starter</artifactId>
        <version>${keycloak.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
  </build>

</project>
```

Since the API gateway is protected with Keycloak in order to protect the backend services, all url patterns except for the health endpoints are secured with Keycloak. The API gateway is developed as a spring boot application. The following java

configuration should be added in order to enable Keycloak security for the API gateway.



```
@Configuration
@ComponentScan(basePackageClasses = KeycloakSecurityComponents.class,excludeFilters = @ComponentScan.Filter(type = FilterTy
               pattern = "org.keycloak.adapters.springsecurity.management.HttpSessionManager"))
@Order(1)
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {

    /**
     * Registers the KeycloakAuthenticationProvider with the authentication manager
     *
     * @param authenticationManagerBuilder
     */
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder authenticationManagerBuilder) {
        KeycloakAuthenticationProvider keycloakAuthenticationProvider = keycloakAuthenticationProvider();
        keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new SimpleAuthorityMapper());
        authenticationManagerBuilder.authenticationProvider(keycloakAuthenticationProvider);
    }

    /**
     * Use Spring boot properties.
     *
     * @return
     */
    @Bean
    public KeycloakSpringBootConfigResolver keycloakSpringBootConfigResolver() {
        return new KeycloakSpringBootConfigResolver();
    }

    @Bean
    @Override
    protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
        return new NullAuthenticatedSessionStrategy();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        super.configure(http);
        http.csrf().disable().cors().and().headers().frameOptions().sameOrigin().and()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
                .authorizeRequests()
                .antMatchers(HttpMethod.GET,  ...antPatterns: "/actuator/**").permitAll()
                .antMatchers(HttpMethod.POST,  ...antPatterns: "/auth/**").permitAll()
                .antMatchers( ...antPatterns: "/**").authenticated();

    }
```

First Keycloak needs to be registered as an authentication provider. Afterwards, the KeycloakSpringBootConfigResolver is registered as a Bean in order for the main Keycloak configuration to be read from the application properties of the Spring Boot application. The overridden configure method defines the security aspects enforced for the incoming HTTP requests. As shown in the configuration it operates in the stateless mode and hence no user sessions are used. GET requests received to the "/actuator" endpoint is allowed without authentication so are the POST requests received to the Authentication server. The "/actuator/**" endpoint is offered by the

Spring actuator project of spring where it allows to monitor and gather metrics about the application. Except for these endpoints, all other url patterns need to be authenticated as defined in the security configuration. In order for the required Keycloak beans to get created in the Spring boot application context, the following properties need to be added to the application.properties file.

```
#Keycloak properties
keycloak.auth-server-url=${KEYCLOAK_SERVER_URL}
keycloak.realm=${KEYCLOAK_REALM}
keycloak.bearer-only=true
keycloak.resource=gatekeeper
keycloak.ssl-required=external
keycloak.confidential-port=0
```

That is all that needs to be done in order to secure the API gateway. Upon successful configuration, only Http requests that include an authenticated bearer token in its header are permitted. The access token received once a user logs in to the system through the protected front end is required to be included in all the http requests sent to consume the protected services. The following properties define the zuul configurations of the service routes. The following configuration disables services from being called from their service name since it could expose the service names making the services vulnerable. Zuul routes are defined for each service with a customized name.

```
# Disable accessing services using service name (i.e. gallery-service).
# They should be only accessed through the path defined below.
zuul.ignored-services=*

# Map paths to services
zuul.routes.device-discovery.path=/discovery/**
zuul.routes.device-discovery.service-id=device-discovery
zuul.routes.auth-service.path=/auth/**
zuul.routes.auth-service.service-id=auth-service
zuul.routes.auth-service.strip-prefix=false

# Exclude authorization from sensitive headers
zuul.routes.auth-service.sensitive-headers=Cookie,Set-Cookie
```

4.3.1 Service Registry

The Authentication and Device Discovery service model includes a service registry or in other terms a naming server in order to reduce the complexity of service identification in the API gateway. In a cloud-native environment, it is often common to see dynamic hostnames and ports. Hence identifying a service by a hardcoded hostname and port defined in a property file is not the most ideal scenario. Even though in the current solution only an authentication and device discovery service is present, it is the intention to add more generic services in the future. Furthermore, depending on the consumption of the authentication and device discovery services, the service instances might increase in order to satisfy the service demands. Hence a service registry seems to be the ideal solution for the API gateway to refer to in order to find the services. Since the whole solution is mainly based on Spring cloud technologies, the Netflix Eureka service which acts as a service registry was used. The Eureka service is a lookup server. All the backend microservices may register with the Eureka server on startup. After the initial registration process, the services can be accessed using the service name provided in the configuration rather than a hostname and port. For a microservice to register in the Eureka server it should declare it as a Eureka Client. This could be done easily by adding the `@EnableEurekaClient` annotation. In each of the microservices that wish to register under the Eureka service registry, it should declare the following application properties.

```
spring.application.name=gatekeeper
eureka.client.service-url.default-zone=http://localhost:8761/eureka/
```

The `spring.application.name` defines the microservice name that it wants to be referred as while the `eureka.client.service-url.default-zone` defines the Eureka server it wishes to register with.

4.4 Liquid QR Code Scanner

Since the content of the QR code generated by the device discovery service is a encoded a JSON String, a custom QR code scanner is required to decode it and execute the expected functionality. The expected functionality upon scanning of the produced QR code is the following.

1. Invoke the endpoint defined in the service model to register a new device with the device information as the request body.
2. Load the specified url with the bearer token.

A simple react application was developed as a qr code scanner including the above functionalities. A device willing to be registered as a new device should first navigate to the Liquid QR code scanner and scan the QR code displayed in the existing device in order to successfully register as a new device. The developed application is mainly based on two react libraries.

1. react-qr-reader - Used for scanning of QR codes
2. react-native-device-info - Used to retrieve device information

CHAPTER 5

EVALUATION

In order to evaluate the developed service model, a proof of concept application was developed using React. Using the proof of concept application we try to demonstrate how application developers can use the Authentication and Device Discovery as a service in their applications. This application consumes the authentication service to protect the application as well as to manage users. Furthermore, this application consumes the device discovery service to enable multiple device usage. For the proof of concept application we have chosen a map application where users can,

1. Log in from one device.
2. Search for a start and a destination to get a route.
3. Share the application with another device with the application state (Start, Destination and the route).

Even though a user login is not a major requirement for a map application, it is integrated in order to exhibit the full feature set offered by the Authentication and Device Discovery service model. The developed proof of concept application demonstrates how the proposed service model can be used in order to adapt liquid properties in an application. The proof of concept application also exhibits the ability of the device discovery service to transport application state from one device to another using the generated QR code. Figure 11 shows the map application developed with React.

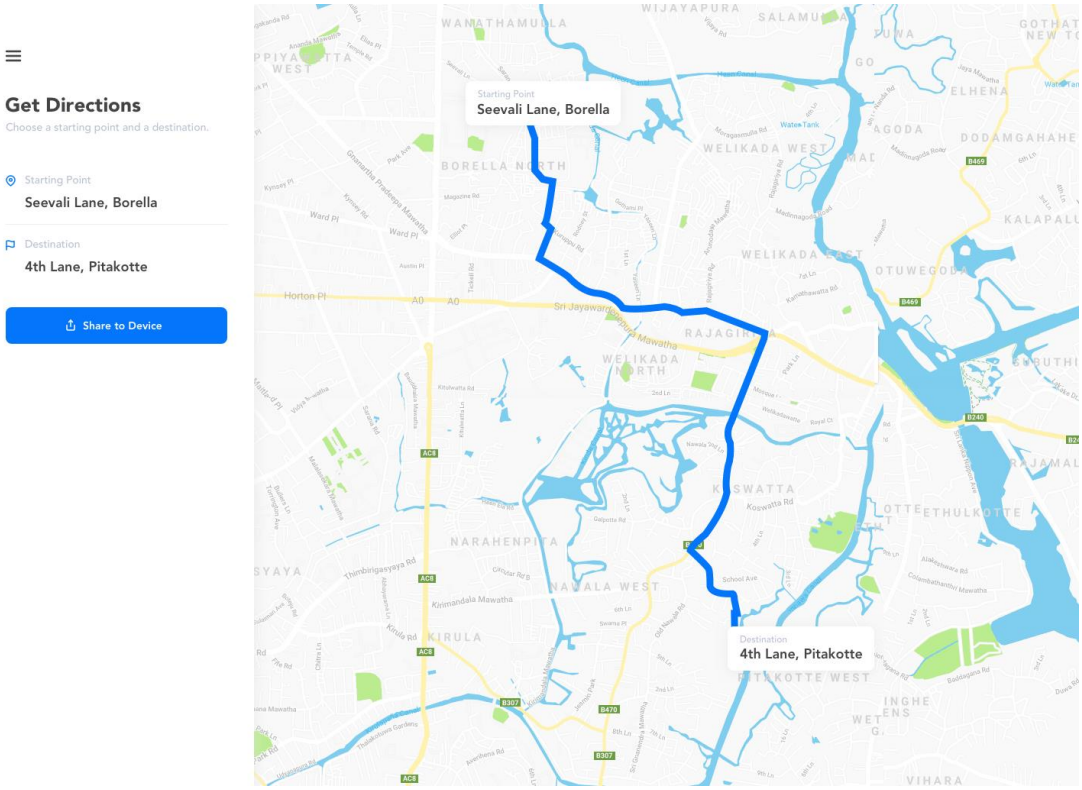


Figure 11 - Proof of Concept Map Application

As shown in the figure, the proof of concept map application simply enables the user to search a location and get directions and shortest path from a source to a destination. In order to access this page the users are required to first authenticate themselves with Keycloak. The default Keycloak login page is shown in Figure 12. As shown in the figure it supports modern authentication features such as Identity brokering. As shown in Figure 13, application developers may configure Identity providers they wish to allow through Keycloak admin console. Once configured those identity providers will be shown in the Login page as shown in Figure 12. Users can then authenticate themselves into the application with the allowed social providers under the “Realm”. As shown in Figure 12, features such as remember me, forgot password and register new users are also available. These features can be configured for each realm (application) through the Keycloak admin console.

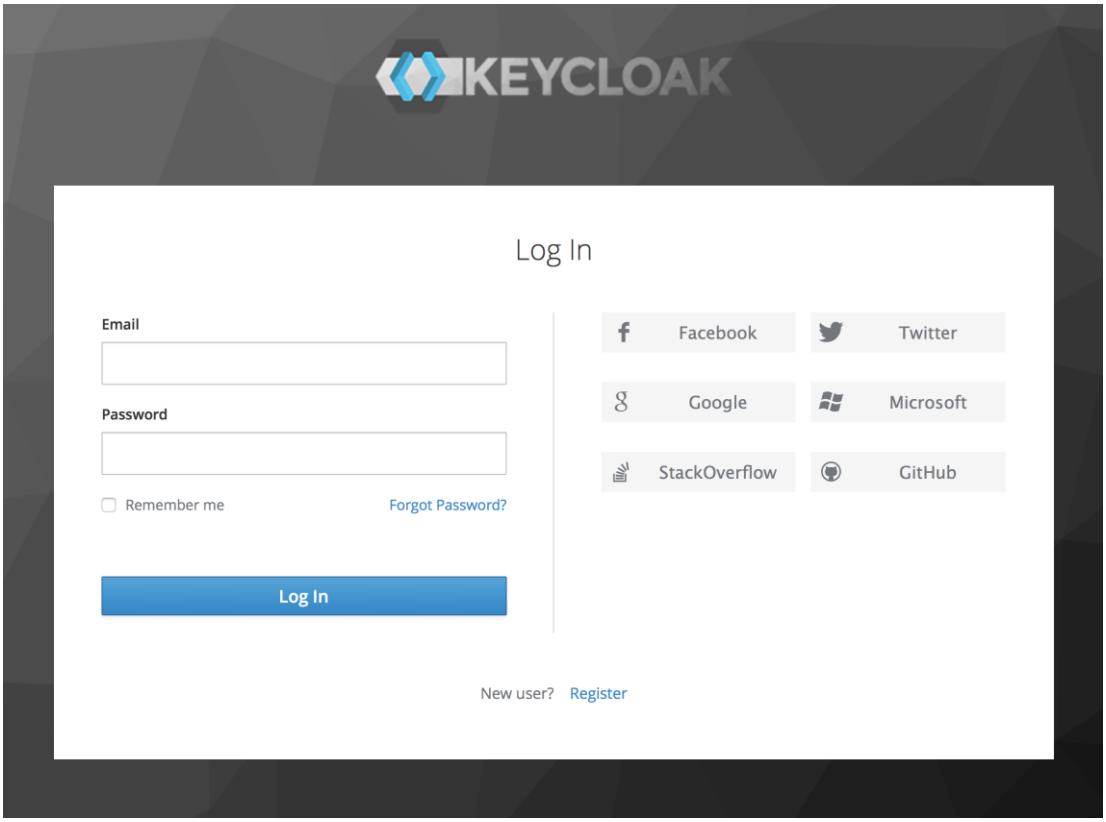


Figure 12- Keycloak Login Page

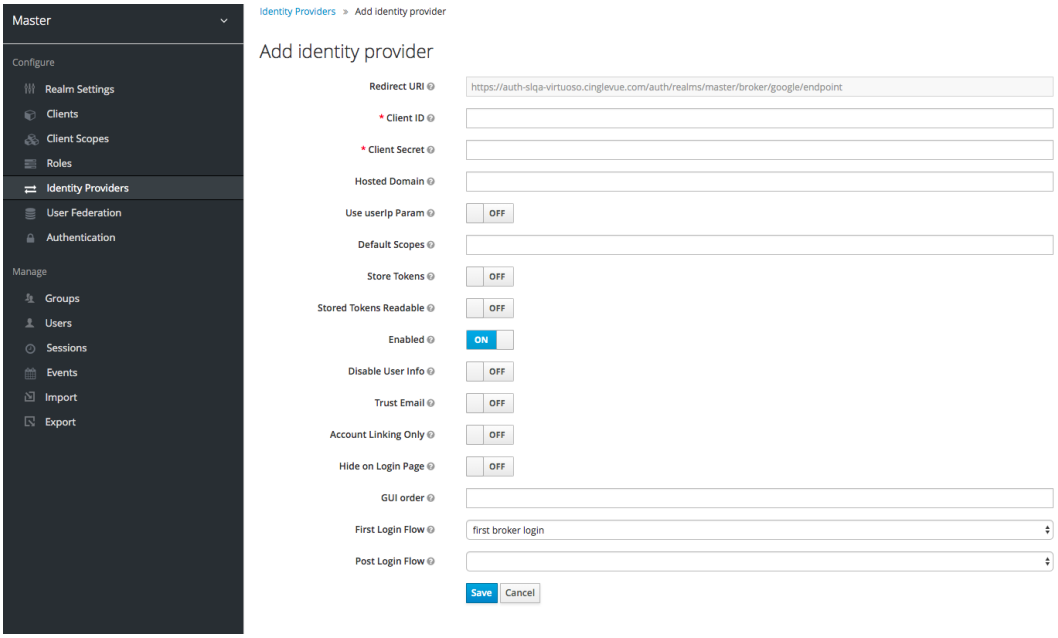


Figure 13- Configuring Identity Providers

The default login page of Keycloak is completely customizable with custom themes. Hence application developers have the freedom to have their own look and feel in the login page to suit their application design. A successfully logged in user will be redirected to the map application. As shown in Figure 11, users may perform the provided functionality of a usual map application. In order to demonstrate the concept of device discovery using the proposed service model, the application has a new button to share the current application to a new device. This is the component in the application which consumes the device discovery service of the proposed service model. The click event of the “Share to Device” button will generate a GET request to the API requesting a QR code to register a new device. The response of the GET request contains the generated QR code in the form of a byte stream. That byte stream is used to display the QR code in the application as shown in Figure 14. Then the new device which the user want to register as his own and continue his work on should then launch the Liquid QR Code scanner and scan this displayed QR code shown in Figure 14. Figure 15 shows the screen where the Liquid QR Code Scanner can be used to scan the QR codes generated by the device discovery service.

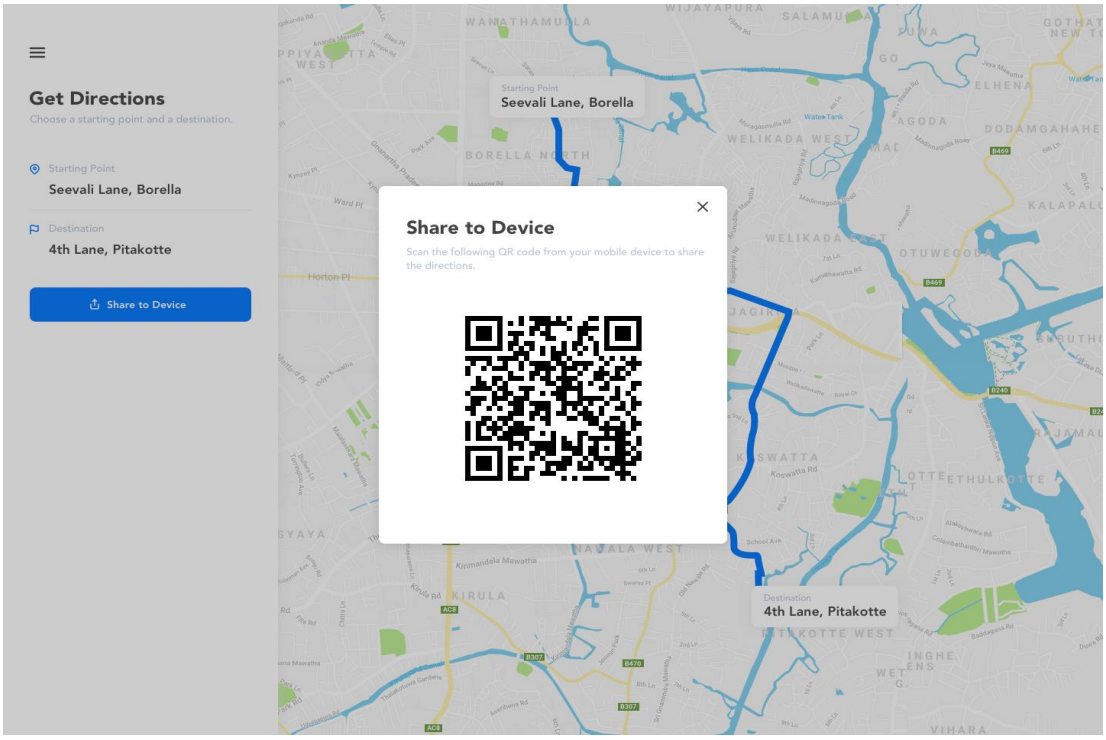


Figure 14- Generated QR code from the Device Discovery Service

Once scanned the Liquid QR Code Scanner performs the following tasks,

- Make a HTTP POST request to the device discovery service in order to register the new device with the acquired device information.
- If the above request returns with a successful response, then redirect the browser to the URL defined in the QR code along with the bearer token in the authentication header.
- Restore any application state variables if available in the metadata of the response.

Once this process is complete, the web browser will load the application in the newly added device while also maintaining the application state where applicable. Figure 16 shows the web browser of the mobile device which scanned the QR code is now redirected to the map application while maintaining the start, destination and route from the previous device. This scenario depicts the sequential device usage.

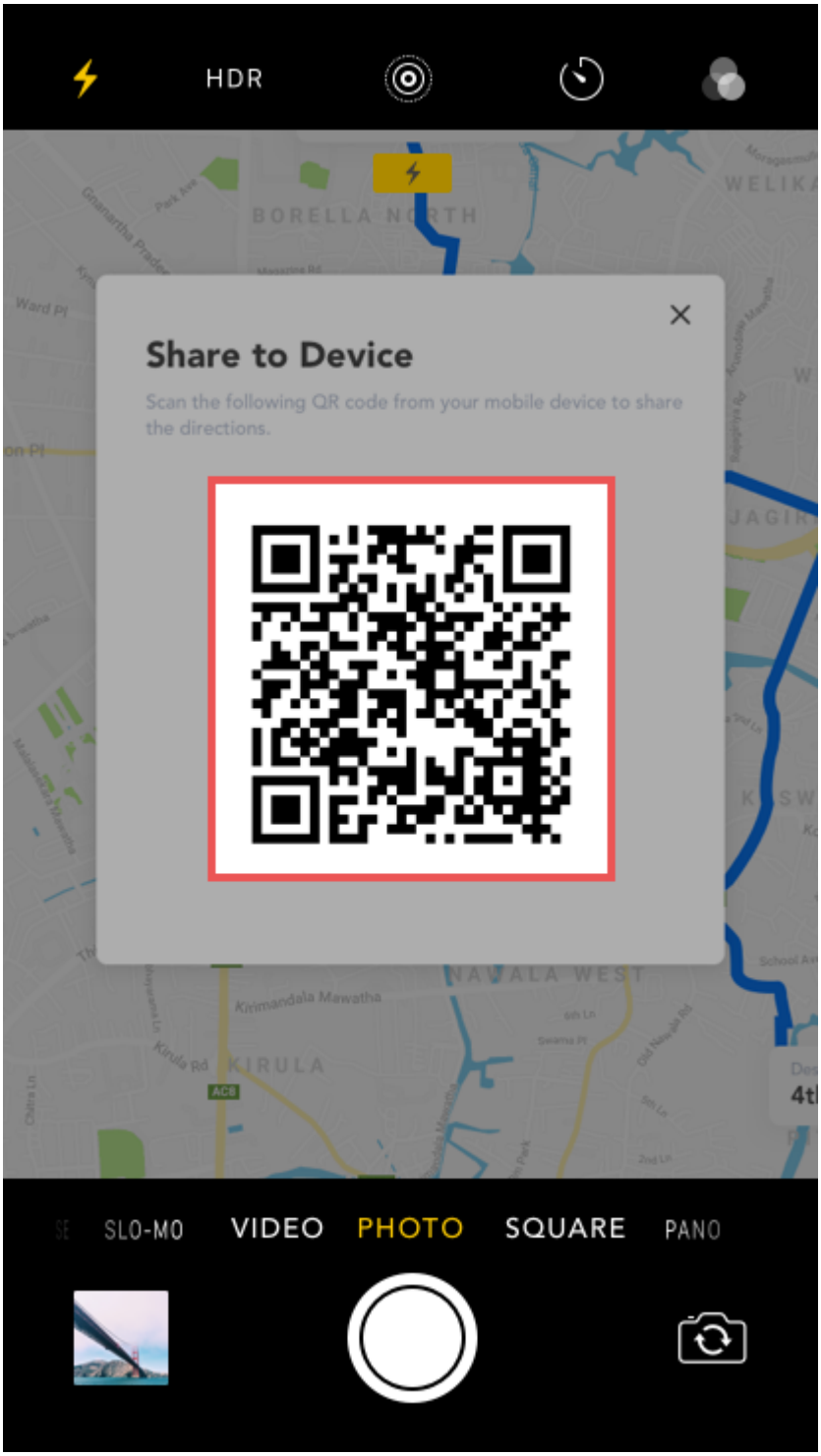


Figure 15 - Liquid QR Code Scanner

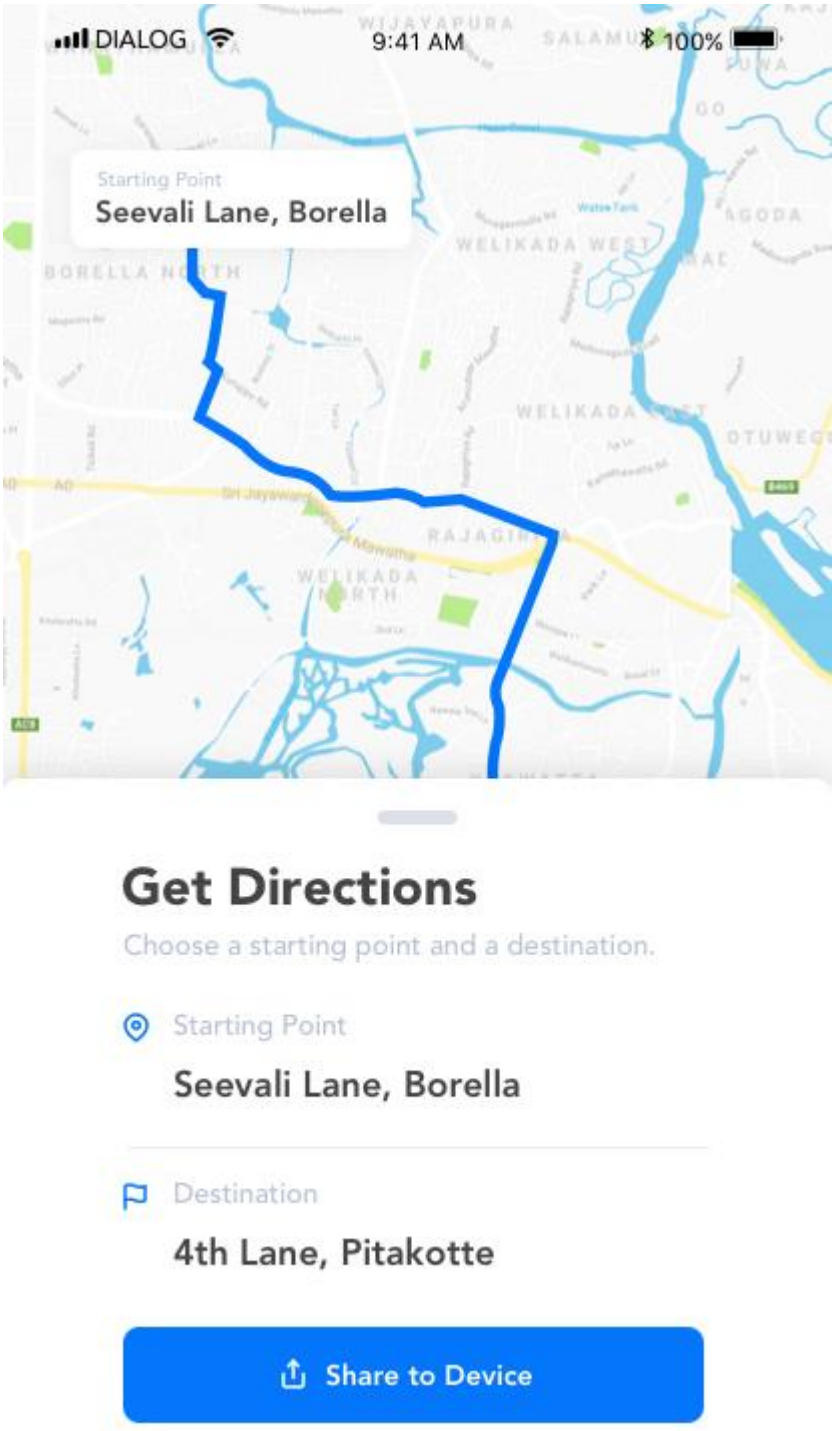


Figure 16 - Application Loaded with state in the target device

This proof of concept application demonstrates the full functionality offered by the Authentication and Device Discovery Service model. This map application is completely based on web technologies and hence is platform independent. The

UI/UX aspects could easily be handled in web applications nowadays with responsive UI components. These design decisions are solely made by application developers and they have a full set of alternatives as discussed in the Liquid Software Design Space [3]. This proof of concept application demonstrates seamless authenticated transition between multiple devices abiding the liquid software manifesto. Furthermore, it demonstrates the ability of the Authentication and Device Discovery service model to ensure security of liquid applications by satisfying the authentication needs. In summary the proof of concept map application that is presented here shows how a generic web application could be transformed in to liquid software application by adapting the proposed Authentication and Device Discovery as a Service.

5.1 Quality of the Service Model

In order to ensure the quality of the services offered by the proposed authentication and device discovery service model SonarQube [27] was used for continuous inspection of code quality. SonarQube is an open source platform that performs static analysis of the source code to detect bugs, vulnerabilities and code smells. When combined with the unit tests SonarQube provides assurance that the source code produced with each commit is of expected quality. SonarQube not only keeps an eye on the health of the application but also tracks down newly introduced issues. This becomes really beneficial for an iterative development strategy hence it can identify bugs in the code as early as possible. Since the service model is still at the preliminary research stage, SonarQube as a service was used with the integration of GitHub repositories. In a production environment, it is ideal to configure SonarQube with a continuous integration engine for scheduled analysis. Each of the project repositories under the service model is scanned with SonarQube to ensure code quality. The Default SonarQube Quality gate was enforced for all the projects. Figure 28 shows the SonarQube dashboard for the gatekeeper project which is the API gateway in the proposed service model. As shown in Figure 17 No bugs, vulnerabilities or code smells were found for the API gateway and the quality gate has been passed.

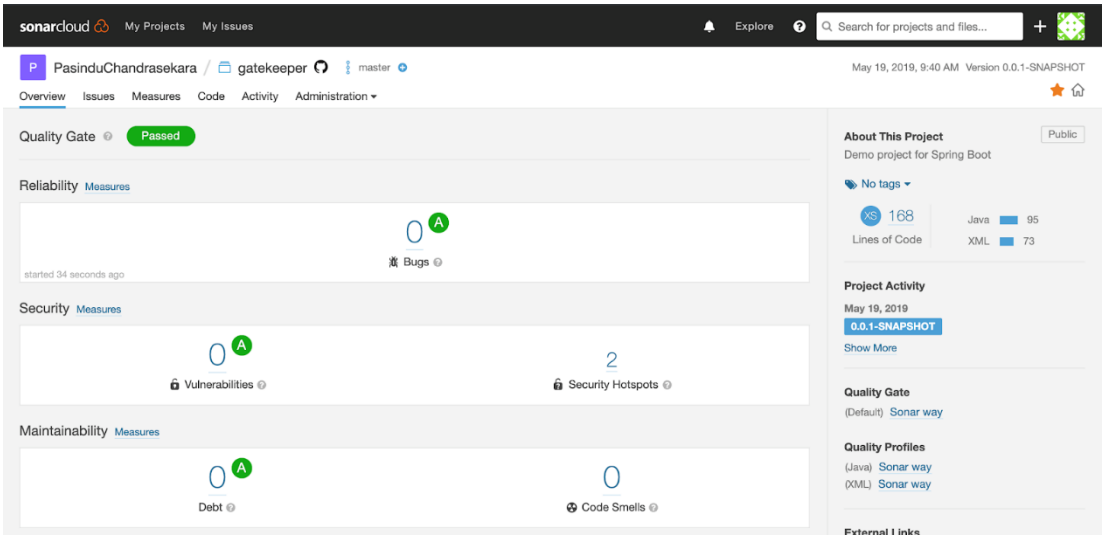


Figure 17- SonarQube Dashboard - GateKeeper

5.2 Concept Evaluation

Having developed a proof of concept application using the proposed service model illustrates the capabilities of the service model. In order to evaluate the functionality that is offered by the service model in real-world applications, the proof of concept application that was developed consuming the service model was evaluated with real users. A group of 25 users representing industry professionals and university students were used for this evaluation process. The following questionnaire was presented to them in order to get their feedback on the user experience. Table 3 contains records of the questions presented in the questionnaire, available answers, the intention of the questions and if the feedback was collected before or after using the application.

Table 3 - Service Endpoints

Question	Available Answers	Intention	Presented Before/After application use
Is it likely that you would want to search a particular route in a map application using your computer, and then transfer it to your mobile and continue the rest from your mobile?	• Most Likely • Likely • Not Applicable	To understand if there's user interest for liquid properties in an application	Before
Were you able to successfully register with the map application?	• Yes • No	Verify the user registration process provided by the service model	After
Were you able to use the provided social login providers such as google and Facebook to login to the application?	• Yes • No	Verify the identity brokering functionality with the configured social providers.	After
Was the "share to device" option provided through	• Yes • No	Background question on the functionality	After

the application useful?		verifying its usage.	
Out of 5 how would you rate the device transition process in terms of continuity?	1 - Interrupted 5 - Seamless Continuation	Verify the functionality introduced by adopting the service model ensures a seamless transition between devices.	After
Out of 5 how easy did you find the device transition process?	1 - Very Hard 5-Completely Hassle-Free	Verify that hassle-free transition between devices is available for applications that adopt the proposed service model.	
On a scale of 5, how accurate was the QR Code Scanning process?	1 - Inaccurate 5 - Very Accurate	Verify the QR Code generated performs the expected functionality offered by the service model.	After
Were you asked to log in to the application from the new device?	• Yes • No	Verify that Single Sign-On is provided through the service model irrespective of the device transition.	After

Did you experience failures when trying to scan the QR code generated using the Liquid QR Code Scanner?	• Very Often • Sometimes • No Failures	To cover the scenario where the QR Code might contain too much information and would not perform the expected functionality during the scanning process.	After
Were the results obtained from the application using one device available in the other after sharing to a new device?	• Yes • No	In order to verify the application state is successfully transferred to the newly added device.	After
On a scale of 5, how much do you prefer to have the "Share to device" feature in all the web applications you use on a regular basis?	1 - Not required 5 - Would love to have it.	To capture how inspired the users are after experiencing the liquid nature offered through the application.	After

Shown below are the visualizations of the summary of the feedback responses that were collected from the user group.

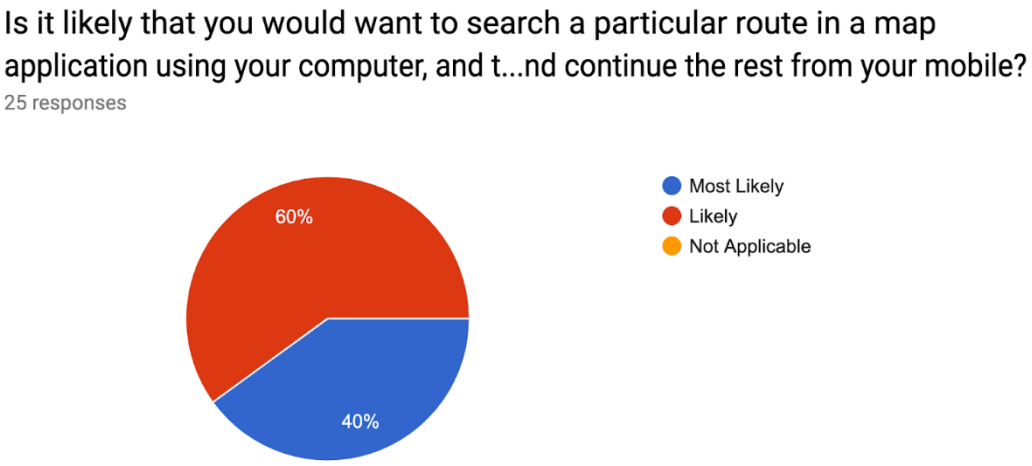


Figure 18 - Feedback - Motivation

Figure 18 shows that out of 25 responses 60% are likely to expect a liquid behaviour from a map application while 40% of the users see it as a most likely feature that they would expect from a map application. Considering this feedback it is evident that all the users have some kind of expectation to have a liquid behaviour in the existing map applications.

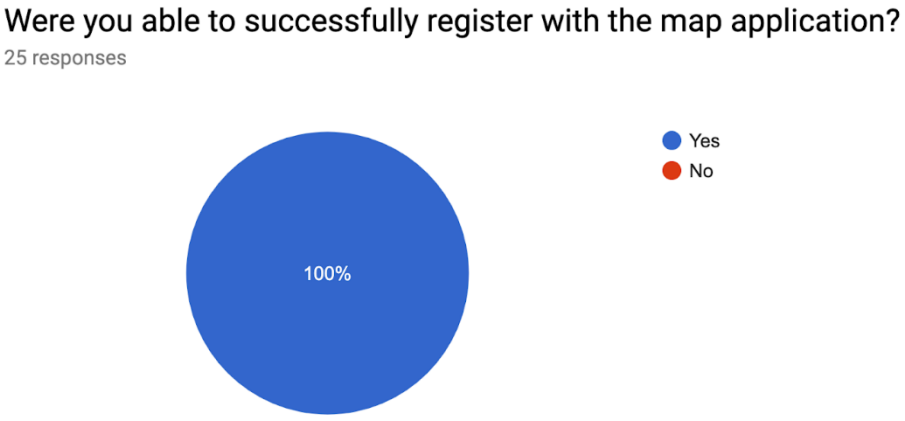


Figure 19 - Feedback - User Registration

Figure 19 shows the user feedback collected on the user registration functionality provided in the proof of concept application. The entire authentication and the user

registration process in the proof of concept were handled using the proposed service model. Since all the users who participated in the evaluation process have been able to successfully register themselves in the application it is evident that the service model can successfully provide user registration functionality to an application.

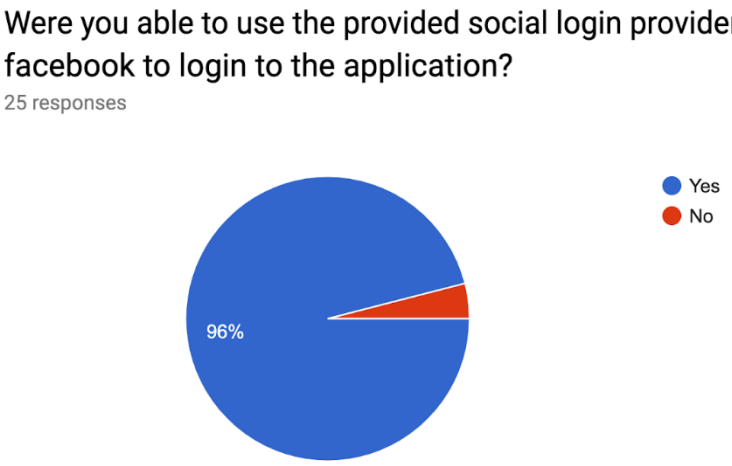


Figure 20 - Feedback - Social Login Providers

Figure 20 shows the feedback responses received with regard to the social login providers that are registered as identity brokers to the application. 96% of the users were able to successfully authenticate themselves into the application using the configured social login providers. The one user who was unable to log in using his social login reported having a problem signing into his social profiles using the proper credentials. In summary 24 out of 25 people were able to successfully authenticate themselves using the provided social login providers.

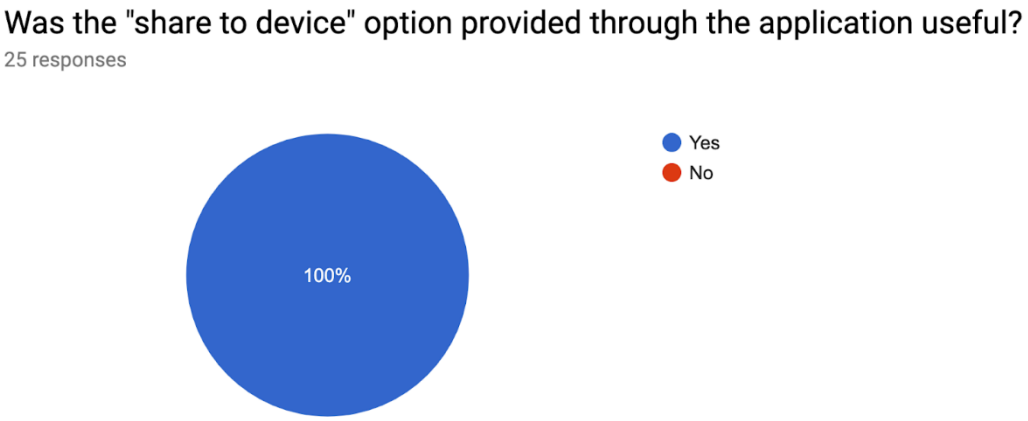


Figure 21 - Feedback - Share to Device Functionality

As shown in Figure 21, 100% of the users agreed that they found the "share to device" option in the application useful. As previously understood with the first feedback question, this also depicts that users are happy to have a feature to share their application sessions with multiple devices. Therefore it can be said that users are happy to experience the liquidity that is provided through the application.

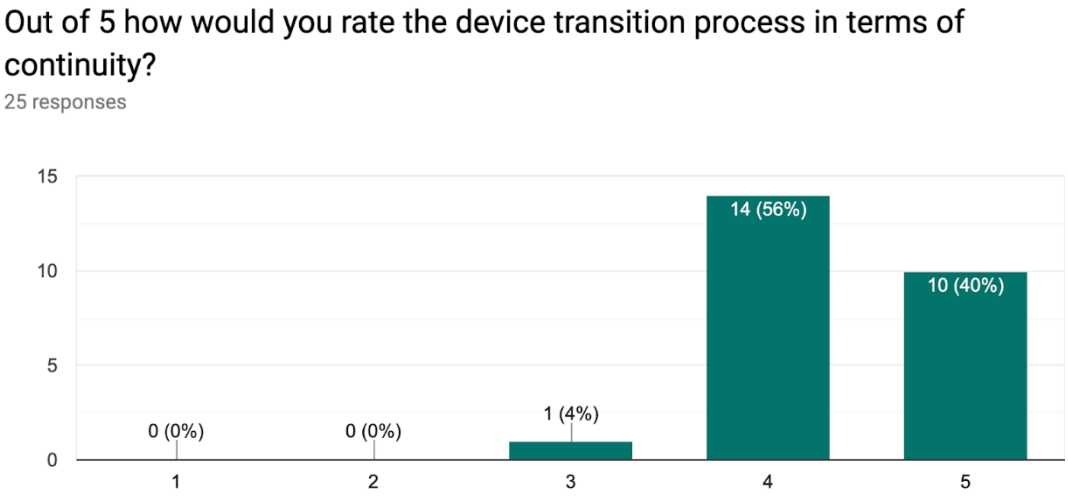


Figure 22 - Feedback - Continuity of the application

Figure 22 shows the feedback responses collected in terms of the continuity experience the users felt during the usage of the application with multiple devices. 96% of the users have responded with above-average continuity experience with ratings of 4(56%), and 5(40%) while 4% of the users felt the continuity is average as they have selected a rating of 3. Majority of the users felt that the application has a really good continuation from one device to another. Users having to launch a specific QR code scanner in order to move from one device to another might be an influential factor in this particular feedback response. But most of the users have felt that the application continuity was really good.

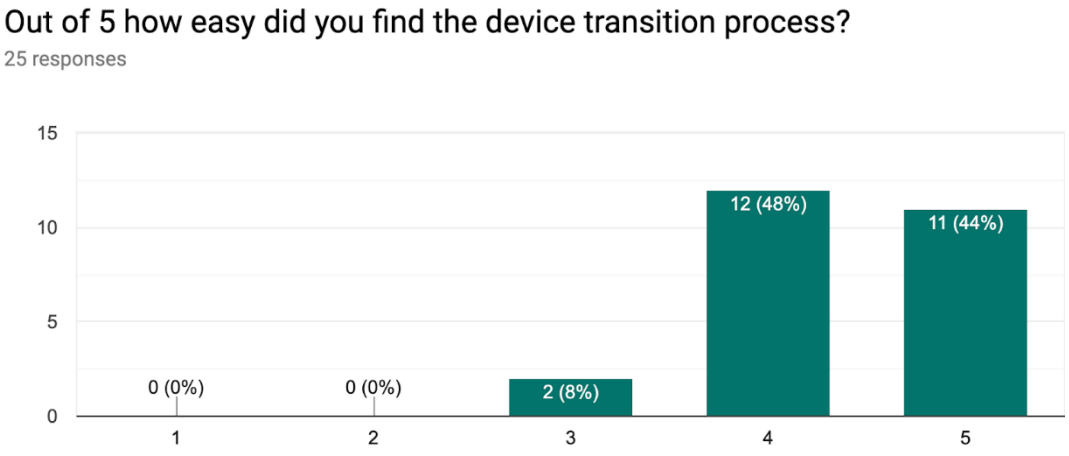


Figure 23 - Feedback - Hassle-Free Transition

Figure 23 shows the feedback responses with regard to the ease of transition between devices. According to the liquid software manifesto, the device transition process needs to be hassle-free as possible. Therefore, the functionality that is provided by the device discovery service of the proposed service needs to be evaluated in that regard. As shown in the summary of the responses, only 2 out of 25 people felt it was at a satisfactory level. All the rest of the users felt the device transition process was above satisfactory. 44% of the users felt the transition process was really easy and hassle-free as possible while another 48% of the users almost agreed to it. Hence it is

evident that the device discovery service, when consumed by an application, can provide a hassle-free transition between devices showcasing liquid behaviour.

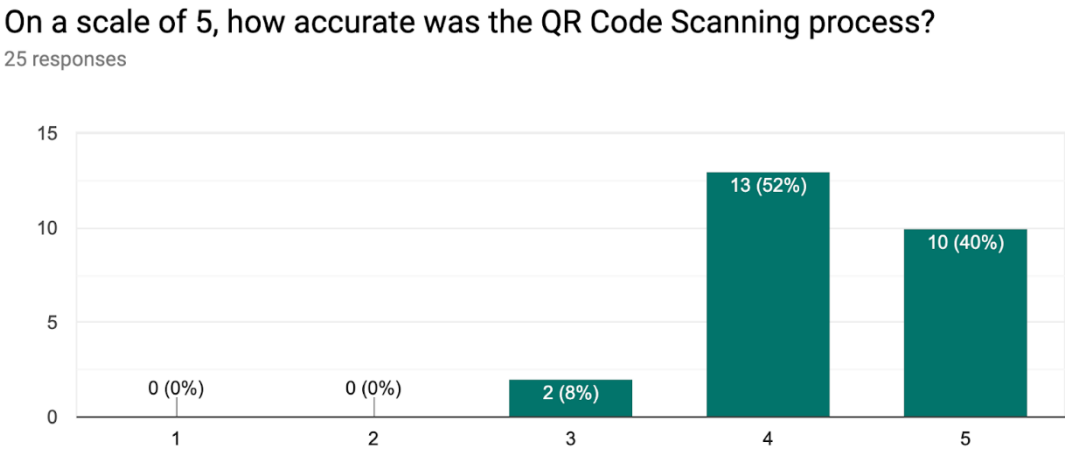


Figure 24 - Feedback - QR Code Scanner Accuracy

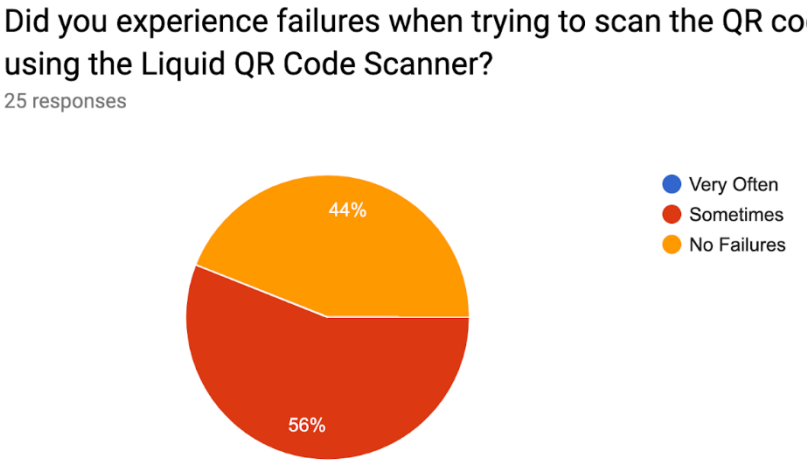


Figure 25 - Feedback - QR Code Scanner Failures

Figures 24 and 25, shows the feedback gathered in terms of the accuracy and the failures of the QR Code scanner as experienced by the users. Some latency was expected during the scanning process of the QR code as the initial phase of the implementation included a considerable amount of data encoded in the QR Code. It

has been identified by the users as well since the QR Code Scanning has sometimes failed for 56% of the users. An alternative approach to including additional data in the QR code has already been identified and will be added to the device discovery service as an improvement.

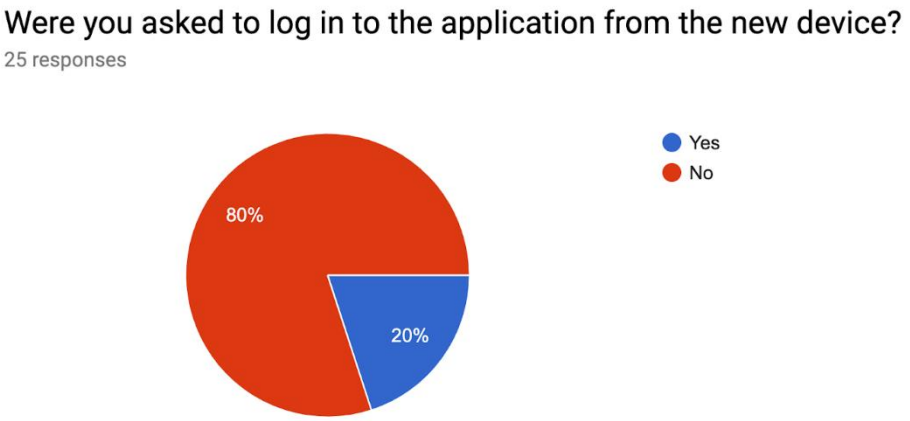


Figure 26 - Feedback - Single Sign-On

Single Sign-On feature is a requirement for a hassle-free device transition. If a user has to authenticate themselves in each and every device that they want to use the application on, then the user experience is affected. In the proposed authentication and device discovery service model single sign-on feature is included depending on a expiration interval which could be configured by the application developers who choose to adopt the service model. As shown in Figure 26, 80% of the users did not require to login again from the new device. The other 20% who had to log in again were understood to have expired authentication tokens meaning they have spent some time going over the expiry interval of the access token that is issued by the authentication server.

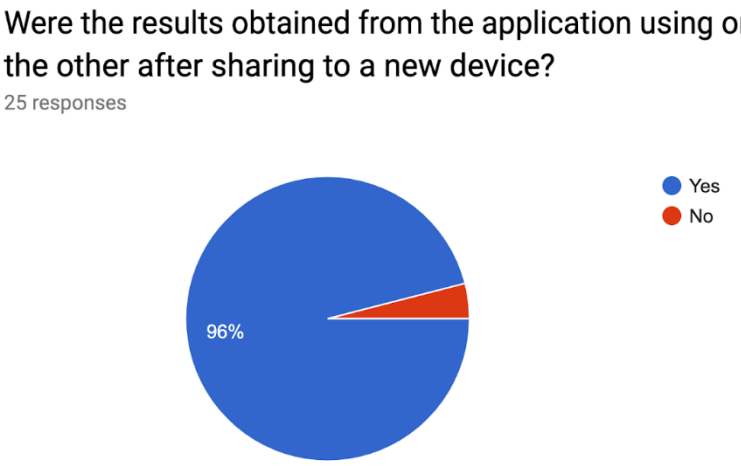


Figure 27 - Feedback - Application state transfer

Figure 27 shows the feedback responses with regard to the application state transfer. In the proof of concept application only the start, destination and the route of the application were treated as the application state that is required to be transferred among devices. Hence as shown in Figure 27, almost all the users have been successful in transferring their application state using the functionality offered by the proposed service model. The reason for one failure that is reported must be due to a mishap during the scanning process of the generated QR code, which was identified earlier as a concern.

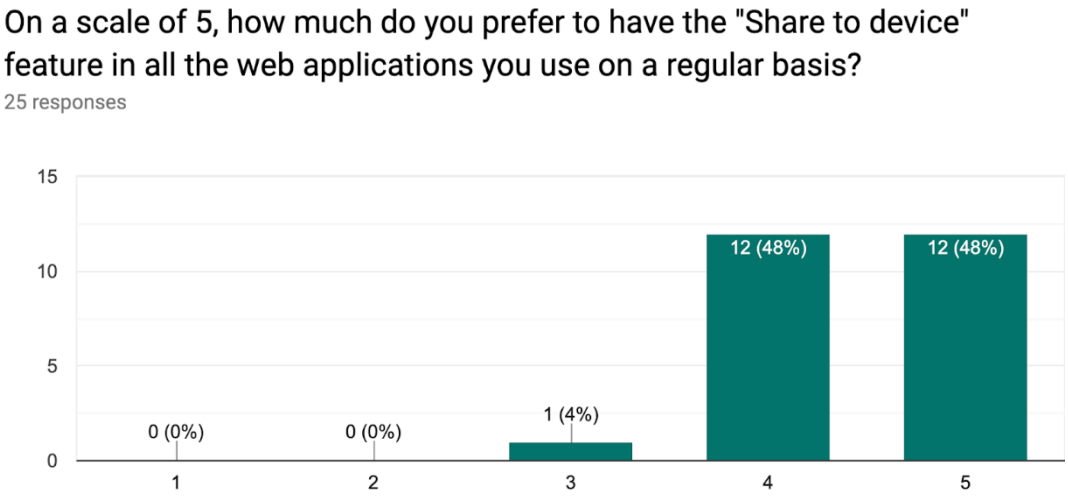


Figure 28 - Feedback - Motivation

Figure 28 shows how much the users expect such functionality offered in the proof of concept application using the proposed service model in all their routine applications. This is a motivational response from the users since almost all of the users have shown great interest in the “share to device functionality”. This is also evidence that users are craving for liquid software applications that suit their modern multi-device ecosystems.

5.3 Limitations

The proposed Authentication and Device Discovery Service model depends on a QR code to support the device registry process. Since the device discovery process needs to be combined with the authentication service, the generated QR code is not just an encoded URL as in the common scenario. Due to the customized nature of the generated QR code, a specific QR code scanner that complies with the same standards is required for a successful device migration. To suffice this requirement the Liquid QR Code Scanner was developed. This would enforce the application user to first navigate to the Liquid QR Code Scanner from the device that they wish to migrate to and scan the generated QR code. The limitation here is the additional step that users need to take during the device transition phase. If it was possible to just scan it from any existing camera application, the scenario would have been ideal. This is a user experience concern that requires further improvement.

Furthermore, the application state synchronization is another concern in liquid software applications. Even though the proposed solution enables the application developers to transfer application state from device to device, it is handled by the metadata stored in the QR code. It is not recommended to have large content in QR codes as it may introduce cross-link mishaps. Therefore the application state that could be transferred from device to device using the current solution would be minimal. Therefore that approach may not be ideal for applications which store bulk of the information in the application state. An alternative approach has already been identified and will be worked on as an improvement.

CHAPTER 6

CONCLUSION

Liquid Software is an essential component in ubiquitous computing. With multiple device ownership becoming the current norm, it is a rising requirement of users to be able to seamlessly migrate from one device to another. But consolidating application security is of utmost importance when it comes to liquid software as compromising one device may lead to the entire application being compromised. Furthermore, application security process should be as hassle-free as possible to the end users. Otherwise it becomes difficult to maintain a seamless transition between devices. Including authentication to secure an application is a commonly seen approach. But if users are required to authenticate themselves to the same application each time they migrate from device to device it disturbs the concept of seamless transition. Therefore it is against the liquid software manifesto.

Having to consider design, development, testing and maintenance of a flexible authentication mechanism for each and every application that is developed is a tedious task. If such functionality is readily available for application developers, it would save them a considerable amount of effort. In a true liquid environment, registration of new devices and authentication should be handled as a combined task. In the proposed service model which is presented in this research, the application developers are provided with the functionality to register new devices along with a proper authentication mechanism. When consumed by an application, the Authentication and Device discovery service handles the authentication of the application and also the device discovery process. When a front end of an application is protected as defined in the model, a user who is authenticated in to the front end of the application doesn't have to repeat himself when he migrate to a different device until the access token received by the authentication service is expired. Migration to other devices is handled by using a QR code generated by the device discovery service. The proposed transition to multiple devices is not only seamless but also transparent since the user devices that a particular user wish to migrate is entirely based on his will. Furthermore, signing off from one device may mean signing off from all devices to handle security. But just in case if a particular user want to unregister a certain device, he could do so by using the functionality provided by the device discovery service. The proposed solution also tries to support application state

transfer to some extent during the device discovery stage. The consumers of the device discovery service may pass on application state in the form of metadata and in return consume them from the registered device. This is not a complete solution to the synchronization of application state but just an additional use case that could be covered by using the proposed service model. In summary Authentication and Device Discovery Service model provides out of the box features to support liquid nature in web applications. A web application which adapts the proposed model is guaranteed to be at least semi liquid according to the liquid software manifesto without requiring additional development effort.

6.1 Future Work

Providing consumption ready services for web applications is a largely developing trend in the software industry. The proposed service model provides Authentication and Device Discovery as a Service for web applications. The intention of this service model is to provide liquid nature to web applications. The existing service model could be further improved by adding more generic services that could support liquid behaviour. One of the identified elements is application state transfer. Even though it is possible to use the current solution to transfer application state in small scale, it will not serve well as the application state continues to scale. Hence a scalable solution is required to support application state transfer. As a future improvement we plan to integrate application state synchronization service in to the proposed service model. Taking out the Keycloak dependency from the service model in order to make it usable with other authentication servers will be another valuable improvement. Upon completion, the service model will be able to provide full liquidity for web applications covering almost all the requirements stated in the liquid software manifesto.

REFERENCES

[1] Taivalsaari, A., Mikkonen, T. and Systä, K. (2014). Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture.

[2] Mikkonen T., Systä K., Pautasso C. (2015) Towards Liquid Web Applications. In: Cimiano P., Frasincar F., Houben GJ., Schwabe D. (eds) Engineering the Web in the Big Data Era. ICWE 2015. Lecture Notes in Computer Science, vol 9114. Springer, Cham

[3] A. Gallidabino et al., "On the Architecture of Liquid Software: Technology Alternatives and Design Space," 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, 2016, pp. 122-127.

[4] Gallidabino, A., Pautasso, C., Mikkonen, T., Systa, K., Voutilainen, J. P., & Taivalsaari, A. (2017). Architecting liquid software. Journal of Web Engineering, 16(5-6), 433-470.

[5] D. Bonetta and C. Pautasso, "An Architectural Style for Liquid Web Services," 2011 Ninth Working IEEE/IFIP Conference on Software Architecture, Boulder, CO, 2011, pp. 232-241.

[6] Gallidabino, Andrea & Pautasso, Cesare. (2016). The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices. 183-186.

[7] Hartman, J.H., Bigot, P.A., Bridges, P.G., Montz, A.B., Piltz, R., Spatscheck, O., Proebsting, T.A., Peterson, L.L., Bavier, A.C.: Joust: A platform for liquid software. IEEE Computer 32(4), 50–56 (1999)".

[8] J. J. Hartman, U. Manber, L. L. Peterson, and T. A. Proebsting, Liquid software: a new paradigm for networked systems. Univ. of Arizona Tech Report TR 96-11, 1996.

[9] Oracle.com. (2017). Sun Ray Products Overview. [Online] Available at: http://www.oracle.com/technetwork/serverstorage/sunrayproducts/overview/index.html [Accessed 10 Aug. 2017].

[10] Apple Support. (2017). Use Continuity to connect your Mac, iPhone, iPad, iPod touch, and Apple Watch. [Online] Available at: https://support.apple.com/en-us/HT204681 [Accessed 12 Aug. 2017].

[11] Bell, K. (2017). Baton promises to be the ultimate Android app switcher. [Online]Mashable.Available at: http://mashable.com/2014/10/27/nextbitbaton-app/ [Accessed 18 Aug. 2017].

[12] Windows Central. (2017). Continuum. [Online] Available at: https://www.windowscentral.com/continuum [Accessed 18 Aug. 2017].

[13] Taivalsaari, A. and Syst, K. (2012). Cloudberry: An HTML5 Cloud Phone Platform for Mobile Devices. IEEE Software, 29(4), pp.40-45.

[14] J. Kuuskeri, J. Lautamäki, and T. Mikkonen, Peer-to-peer collaboration in the Lively Kernel. Proc. 25th ACM Symposium on Applied Computing, 2010, pp. 812-817.

[15] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, Web browser as an application platform: The Lively Kernel experience. Sun Microsystems Laboratories Tech Report TR-2008-175, 2008.

[16] K. Systä, L. Järvenpää, and T. Mikkonen, HTML5 agents – mobile agents for the web. Proc. International Conference on Web Information Systems and Technologies 2013 (WebIST'13, Aachen, Germany, May 8-10), 2013, pp. 37-44

[17] Bourges-Waldegg, D., Duponchel, Y., Graf, M., and Moser, M. (2005) The fluid computing middleware: Bringing application fluidity to the mobile internet. IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05), pp. 54–63, IEEE.

[18] Palmer, T. D. and Fields, N. A. (1994) Computer supported cooperative work. Computer, 27, 15–17.

[19] Grundy, J., Wang, X., and Hosking, J. (2002) Building multi-device, component-based, thin-client groupware: Issues and experiences. Australian Computer Science Communications, vol. 24, pp. 71–80, Australian Computer Society, Inc.

[20] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen, The Lively Kernel – a self-supporting system on a web page. Proc. Workshop on Self-Sustaining Systems (S3'2008, Potsdam, Germany, May 15-16, 2008), LNCS5146, Springer-Verlag, 2008, pp. 31-50.

[21] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, Web browser as an application platform: The Lively Kernel experience. Sun Microsystems Laboratories Tech Report TR-2008-175, 2008.

[22] "Spring Projects", *Spring.io*, 2019. [Online]. Available: https://spring.io/projects/spring-cloud. [Accessed: 06- Mar- 2019].

[23] K. Team, "Keycloak - Documentation", *Keycloak.org*, 2019. [Online]. Available: https://www.keycloak.org/documentation.html. [Accessed: 08- Mar- 2019].

[24] "React – A JavaScript library for building user interfaces", *Reactjs.org*, 2019. [Online]. Available: https://reactjs.org/. [Accessed: 20- Mar- 2019].

[25] "OAuth 2.0 — OAuth", *Oauth.net*, 2019. [Online]. Available: https://oauth.net/2/. [Accessed: 01- Mar- 2019].

[26] "OAuth 2.0 Grant Types", *Oauth.net*, 2019. [Online]. Available: https://oauth.net/2/grant-types/. [Accessed: 01- Mar- 2019].

[27] "Continuous Inspection | SonarQube", *Sonarqube.org*, 2019. [Online]. Available: https://www.sonarqube.org/. [Accessed: 13- May- 2019].

[28] "Yjs", *Y-js.org*, 2019. [Online]. Available: http://y-js.org/. [Accessed: 09- May- 2019].

[29] D. HAKOBYAN, "Authentication and Authorization Systems in Cloud Environments", Stockholm, Sweden, 2012.

[30] Openid.net. (2019). *OpenID Connect | OpenID*. [Online] Available at: https://openid.net/connect/ [Accessed 13 May 2019].