|  |
| --- |
| COMP4906 HONOURS THESIS: SKYDOT |
| Final Report |

|  |
| --- |
| Lexi Brown - 100956208 |

Table of Contents

[Summary 2](#_Toc512192245)

[1 Introduction 5](#_Toc512192246)

[1.1 Problem 5](#_Toc512192247)

[1.2 Motivation 6](#_Toc512192248)

[1.3 Project Goals 6](#_Toc512192249)

[1.4 Proposal Objectives 7](#_Toc512192250)

[1.5 Possible Features 7](#_Toc512192251)

[1.6 Technology and Equipment Requirements 8](#_Toc512192252)

[2 Background 9](#_Toc512192253)

[2.1 Architecture 9](#_Toc512192254)

[2.1.1 Microservice 9](#_Toc512192255)

[2.1.2 Layered 9](#_Toc512192256)

[2.2 Technologies 9](#_Toc512192257)

[2.2.1 Azure (AKS) 9](#_Toc512192258)

[2.2.2 Kubernetes 9](#_Toc512192259)

[2.2.3 Docker 9](#_Toc512192260)

[2.3 Alternatives 9](#_Toc512192261)

[3 Solution 10](#_Toc512192262)

[3.1 Client 10](#_Toc512192263)

[3.2 Skydot 10](#_Toc512192264)

[3.2.1 Micro-Apps 11](#_Toc512192265)

[3.2.2 Micro-Services 14](#_Toc512192266)

[3.2.3 Host Gateway 16](#_Toc512192267)

[3.3 Backend 18](#_Toc512192268)

[3.4 DevOps 20](#_Toc512192269)

[3.4.1 Dashboard 20](#_Toc512192270)

[3.4.2 Logging 21](#_Toc512192271)

[3.4.3 Monitoring 22](#_Toc512192272)

[4 Results/Validation 23](#_Toc512192273)

[Project Goals 23](#_Toc512192274)

[5 Conclusion 24](#_Toc512192275)

[5.1 Future Work 24](#_Toc512192276)

[Additional Figures 26](#_Toc512192277)

[References 27](#_Toc512192278)

# Summary

The rise of microservices has been a remarkable advancement in application development and deployment. With microservices, an application is developed or refactored into individual services that have the capability to communicate with one another through a common template, for instance APIs. Each service is self-contained, manages its own data storage and can be updated independently of other services. Moving to a microservice-based approach makes application development faster and easier to manage; requiring fewer people to develop and maintain the system. A system designed as a collection of microservices is easier to run on multiple servers and, in the case of this project, multiple cloud environments with load balancing. This allows for better handling of demand spikes and of slower increases in demand over time while reducing downtime caused by hardware or software problems.

Microservices are a critical part of the paradigm shift occurring in the way applications are being built. Agile development techniques, the transition from on-premise to cloud, DevOps culture, continuous integration and continuous deployment (CICD), and containerization of applications all work alongside microservices to revolutionize application development and delivery.

In this report, I cover information relevant to implementing microservices through Skydot and the work that went into finding the best way to model Skydot if it was to be implemented in industry. The sections within this report are:

**Introduction** – A simple and clear introduction to Skydot.

1. **Problem** – An outline of the problems Skydot will be addressing.
2. **Motivation** – What has inspired and motivated me to build Skydot.
3. **Project Goals** – A list of all the goals Skydot aims to accomplish.
4. **Proposal Objectives** – Objectives set by this proposal to be completed in the implementation of Skydot.
5. **Possible Features** – A list of possible features that may be completed in the implementation of Skydot.
6. **Technology and Equipment Requirements** – A list of technology and equipment required to complete the project.

**Background** – A summary of all the background research completed in preparation of building Skydot.

1. **Architecture** – A simple outline to the architecture used by Skydot. This includes microservice and layered architecture styles.
2. **Technologies** – A simple introduction to technologies utilized by Skydot.

**Solution** – A summary of how the whole project was built from top to bottom.

1. **Client** – An overview on the web and mobile client solutions.
2. **Skydot** – An in-depth look at how Skydot was built and the capabilities of the solution.
3. **Backend** – An outline of how the backend was put together to represent older and newer technologies.
4. **Completed Objectives** – An overview of all completed features.

**Results/Validation** – An in-depth report of all results and findings involved in analyzing Skydot. And an analysis of key components that will show off Skydots abilities.

**Conclusion** – A summary of all the aspects of the project.

1. **Future Work** – An analysis of how Skydot could be improved in the future.

Every section pertains to the construction of Skydot and meeting the goals outlined by the project. I hope you find every section worthwhile and inspirational to your own utilization of Skydot and cloud-based microservice architecture.

# 1 Introduction

Skydot is a cloud-based architecture that allows companies to minimize the cost of cloud and on-premise services and provides an environment where developers can utilize any language that best suits their needs and/or skills. This is achieved by utilizing a universal REST API and auto scaling services and apps. Skydot also fronts backend services, databases and other resources via a REST translation layer. This way backend services won’t have to change to adopt new technologies and new services won’t have to accommodate for old technologies. The project will be presented as a mobile banking service providing data for Android, Web and, if time permits, iPhone mobile application.

## 1.1 Problem

Skydot will be addressing the high cost of maintaining on-premise technology and the hidden transition costs to cloud based services while maximizing the productivity of software development. These problems encompass the following issues in today’s industry:

* On-premise technology costs are very high
* Maintenance costs increase as hardware gets older and therefore must be replaced
* Many people are needed to manage the infrastructure of on-premise technologies
* Disaster recovery sites are needed to reduce risk and are costly to maintain
* Cloud resources can be expensive if not handled efficiently
* Incorporating new technologies while maintaining old software frameworks can become unmanageable and can cause licensing and compatible issues
* Lack of collaboration between teams causes duplication of code and effort, and risks reduction of data integrity

## 1.2 Motivation

The inspiration for this project was ignited at a company I worked with previously who wanted to move to cloud based services. The industry was, and still is, moving in the direction of cloud-based technologies since it can be cost effective, and forward-thinking companies want to stay ahead on the latest technologies. In the end, the company decided upon out-of-the-box software that does much of what I’ve outlined for this project but is more limiting and costly. I believe there is a cheaper, more efficient and more inclusive way of utilizing cloud services. Many frameworks cost from thousands to millions, depending on the needs of the company purchasing the product, and only provide a limited amount of compatible languages and frameworks that developers can use.

## 1.3 Project Goals

1. Decrease the number of people needed to maintain software and the cost of maintaining that software.
2. Increase code integrity and decrease code development and integration time between teams.
3. Counter long, multi-step manual deployment with simple, quick, autonomous cloud deployment.
4. Handle cloud resources as efficiently as possible.
5. Provide a common point of access for client applications.
6. Create a layered, microservice framework that separate client applications from common services.
7. Provide a common point of access to host services and data.
8. Utilize Skydot as the server side of a mobile banking application to present the capabilities of the project.

## 1.4 Proposal Objectives

1. Set up a Kubernetes application container that wraps the entire project and is used for deployments.
2. Utilize docker to wrap micro-apps and micro-services for deployment within Kubernetes
3. Establish an API gateway in Kubernetes through which micro-apps can register and client applications can send requests.
4. Establish a service gateway through Kubernetes that allows micro-apps to make REST requests to micro-services.
5. Optionally establish a host gateway that provides a REST API for micro-services to access backend services. The backend consists of REST and SOAP services.
6. Build an authentication database server to generate and keep tokens for client application requests.
7. Build backend services for authentication, account information, currency conversion and bill payments. At least one WSDL service must be provided.
8. Provide micro-services in Java, Python and C++. Micro-services coverage: Authentication, Account summary and details, Transfers, Bill payment.
9. Build front end application in Kotlin for Android and in ReactJS for web to display services.

## 1.5 Possible Features

1. DevOps: Dashboard, health checking, logging, monitoring, continuous integration and continuous development (CICD).
2. Populate a string database with error and warning messages (en\_CA and fr\_CA).
3. Build an iPhone and/or tablet application.
4. Integrate oAuth2 and LDAP authentication.
5. Use Swagger for design and documentation.

## 1.6 Technology and Equipment Requirements

* Microsoft Azure
* Android device(s)
* ReactJS
* Android Studio
* Kubernetes
* Docker
* Minikube

# 2 Background

## 2.1 Architecture

A section on how the architecture works. This includes microservice and layered architecture.

### 2.1.1 Microservice

An outline of Skydots microservice architecture

### 2.1.2 Layered

An outline of Skydots layered architecture

## 2.2 Technologies

### 2.2.1 Azure (AKS)

A section on how Azure AKS works

### 2.2.2 Kubernetes

A section on how Kubernetes works

### 2.2.3 Docker

A section on how docker and docker hub work

## 2.3 Alternatives

# 3 Solution

## 3.1 Client

There are two client applications that were made for this project: an Android application and a web application. The Android application was built in both Java and Kotlin while the web application was built in JavaScript. The significance behind building two client applications is to show how two projects using the same services would work within Skydot’s architecture. Each client application has a micro-app built specifically for it. The mobile micro-app services the Android application. It returns data differently than the web micro-app, which services the web application, because a mobile device can’t display as much information on one page as a web application in a browser can. For example, on a bill payee call, which would return a list of bill payees one can use for a bill payment, the Android application provides search functionality for a payee since displaying the entire list of payees takes up too much screen space and would be tedious to scroll through. Whereas the web application can display the entire list, while also providing search functionality, because there is more screen real estate to work with. So, in terms of each micro-app, the mobile micro-app would provide a bill payee search endpoint and the web micro-app would provide a bill payee search and bill payee get all endpoint.

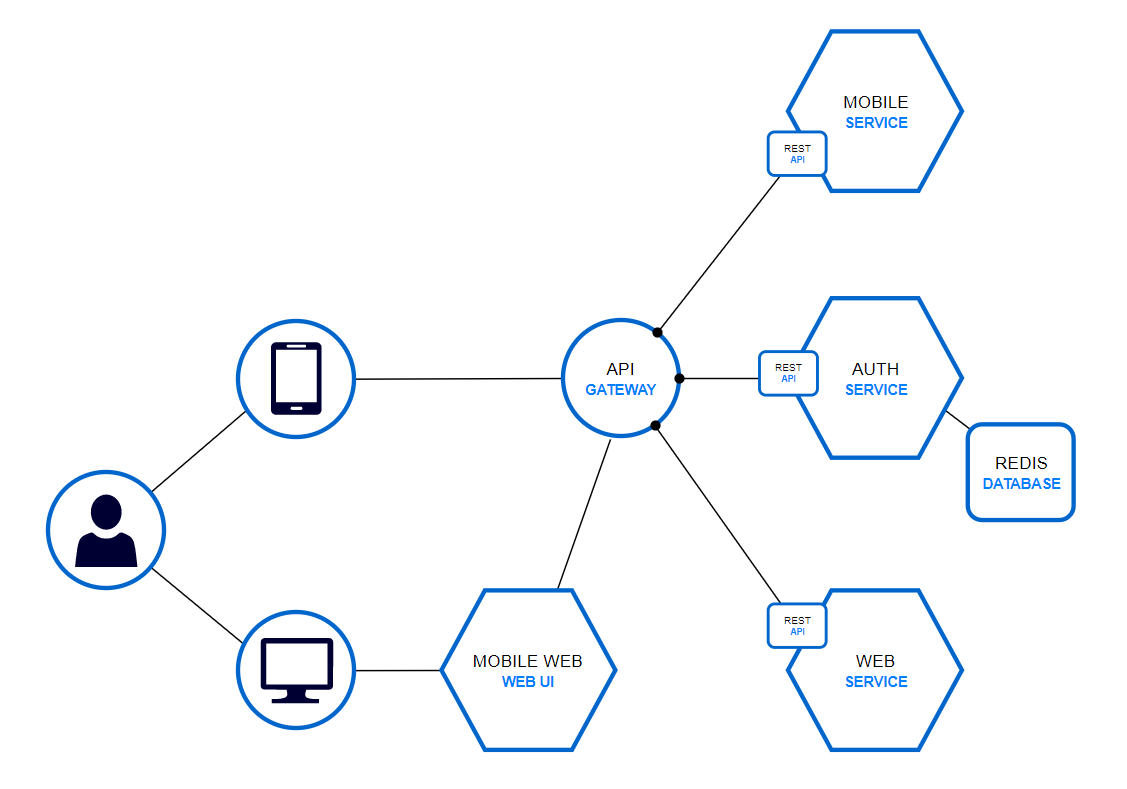
## 3.2 Skydot

Skydot is comprised of three sections: the micro-apps, the micro-services and the host gateway. These three sections are essential to the project as they maintain access control, business logic and data translation. The micro-apps layer is the only one that is publicly accessible while the host gateway and micro-services support the micro-apps from within Skydot’s private virtual network. Although all three layers can access the internet, if need be, only the host gateway can access private backend data services and storages that provide access to secure client information. The following sub-sections further outline each layer.

### 3.2.1 Micro-Apps

The micro-apps are the application layer of Skydots architecture. They sit between the presentation layer and the service layer. Each micro-app is made by a team in service to usually one client application within the presentation layer. In terms of this project, this is represented with the web and mobile micro-apps. The mobile micro-app tailors to the needs of mobile client applications while the web micro-app works with web applications. Both micro-apps require the same backend services but differ is things like request parameters, return types, amount of information returned and development language.

The exception to this template is the authentication micro-app. The authentication (auth) micro-app is not specific to any client. The auth micro-app controls access to all services through token validation; it acts as the central hub for authentication. All clients must login through the auth micro-app to receive a token that will allow them to access other micro-apps. A request to the mobile or web micro-app will be rejected if there isn’t a token in the request. All micro-apps must then verify if the token is valid by requesting verification from the authentication micro-app.



Since the micro-apps are running within the same cluster, the micro-apps can use service discovery to quickly access the auth micro-app without having to know the auth micro-apps internal cluster IP address or having to make a request out of the cluster to the auth micro-apps external IP or DNS. All micro-apps can access the auth micro-app at ‘auth-app’, which is the service name I specified in the YAML definition. This ensures that each micro-app doesn’t need to know the specifics of other micro-app (i.e., cluster IP, number of pods, node location, etc) to make a request to it. If the auth micro-app validates the token the user is valid, and request can go through, provided the any other required parameters are present.

|  |  |  |  |
| --- | --- | --- | --- |
| Micro-App | Service Name | Language | Used By |
| Authentication | auth-app | Python | All clients and micro-apps |
| Mobile | mobile-app | Java, Kotlin | Mobile client |
| Web | web-app | JavaScript | Web client |

**Authentication Micro-App –** This micro-app handles everything to do with authentication. It is developed in python and utilizes a redis cache to store logged in users. There are four endpoints that the app handles: /auth/login, /auth/logout, /auth/verify and /auth/user. These endpoints allow clients to login, and receive a session token, and logout. And they allow micro-apps to verify client session tokens and retrieve the user id that is encrypted into the token. No micro-app, other than the auth micro-app, knows how to encrypt or decrypt session tokens. This ensures that all login tokens are handled by one application and that every client logs in the same way.

**Mobile Micro-App –** This micro-app handles all service requests a mobile client can make except logging in. Here is a list of all the endpoints it provides and what micro-service each endpoint goes to in default mode.

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Micro-Service |
| /account | GET | None |
| /account | POST | account-summary-service-python |
| /account/details | POST | account-details-service-python |
| /bill | GET | None |
| /bill | POST | bill-payment-service-javascript |
| /bill/payee | POST | bill-payee-service-javascript |
| /bill/payee/search | POST | bill-payee-service-javascript |
| /transfer | GET | None |
| /transfer | POST | transfer-service-java |
| /user | GET | None |
| /user/create | POST | create-service-cplus |
| /user/delete | POST | delete-service-cplus |

**Web Micro-App –** This micro-app handles all service requests a web client can make except logging in. Here is a list of all the endpoints it provides and what micro-service each endpoint goes to in default mode.

|  |  |  |
| --- | --- | --- |
| Endpoint | Method | Micro-Service |
| /account | GET | None |
| /account | POST | account-summary-service-python |
| /account/details | POST | account-details-service-python |
| /bill | GET | None |
| /bill | POST | bill-payment-service-cplus |
| /bill/payee | POST | bill-payee-service-java |
| /bill/payee/search | POST | bill-payee-service-java |
| /transfer | GET | None |
| /transfer | POST | transfer-service- javascript |
| /user | GET | None |
| /user/create | POST | create-service- python |
| /user/delete | POST | delete-service- python |

Each micro-app has its own external IP and DNS so that, even if the micro-apps have the same endpoints, each request will go to the correct micro-app. The DNS are as follows:

**Auth** – auth-skydot.<azure-dns>.com

**Mobile** – mobile-skydot.<azure-dns>.com

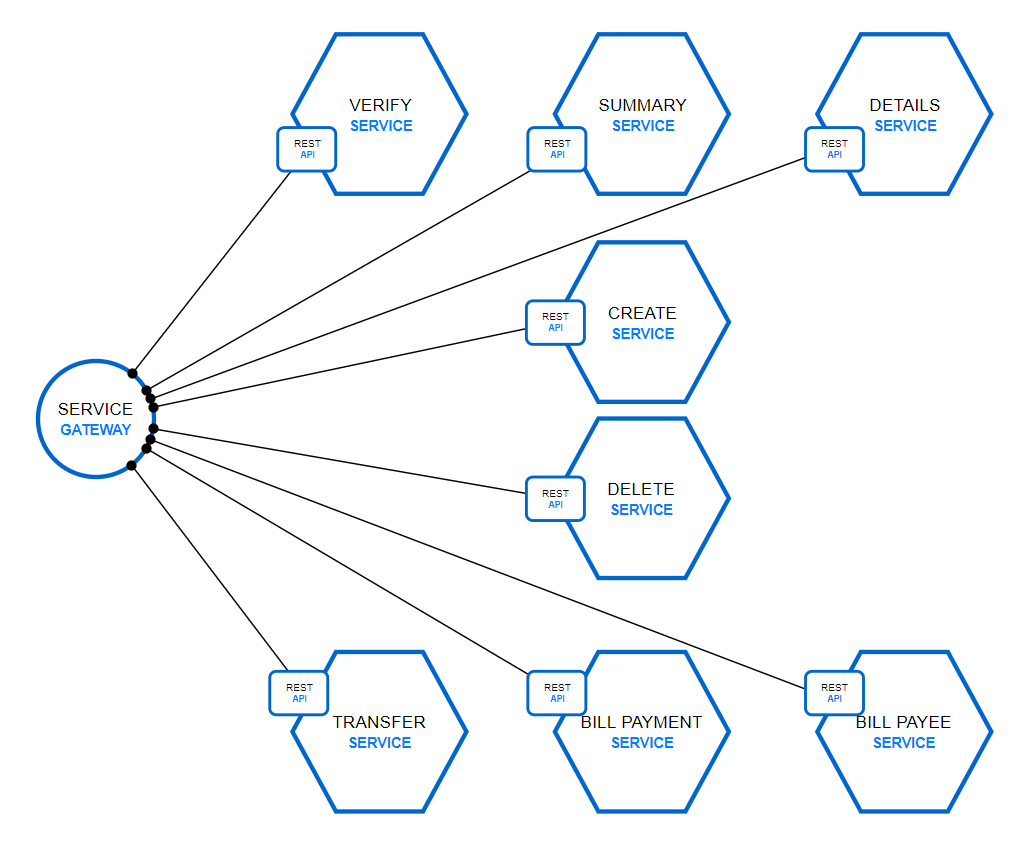
**Web** - web-skydot.<azure-dns>.com

As the banking application part of this project is just for demonstration purposes, I have used the default DNS Azure provides for public IP addresses. However, it is possible to use ones own custom domain if they own that domain.

### 3.2.2 Micro-Services

The micro-services are within Skydots service layer. This layer is between the application and data layers. Micro-services are common services used by micro-apps in the application layer. They typically provide only one service. In this project I have five categories of services and a total of eight services between them. A list of categorized services follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Authentication | User | Account | Transfers | Bills |
| - verify | - create  - delete | - account summary  - account details | - transfer | - bill payment  - bill payee |



However, there are total of nineteen micro-services deployed in Skydot. This is because micro-services can be developed in almost any language. If the micro-service development language can provide a REST api then it can be used. So, I have set up duplicate services in different languages. This does change service discovery a bit since you cannot deploy multiple services with the same name. Therefore, I have a simple naming format for micro-services which is as follows:

<service name>-service-<language>

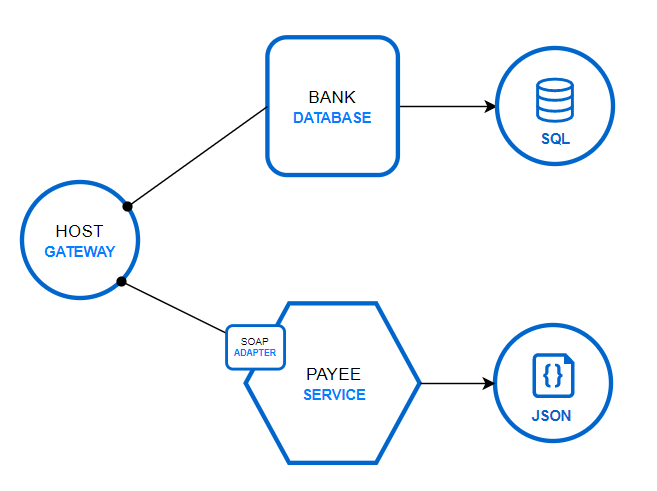
This format depicts the service name needed in service discovery. All micro-services are private and can only be accessed within the clusters’ private virtual network. Although each micro-service has a cluster IP, that IP is dynamic and will change when a new pod of that service is created. So, the service name in my specified format is used, which is defined in the YAML for each service. Below is a list of all micro-services:

|  |  |  |  |
| --- | --- | --- | --- |
| Service | Service Name | Endpoint(s) | Languages |
| Authentication | verify | /auth | C++\*, Python |
| Account Summary | account-summary | /account/summary | Python |
| Account Details | account-details | /account/details | Python |
| User Create | create | /create | C++, Python |
| User Delete | delete | /delete | C++, Python |
| Transfers | transfer | /transfer | C++, Java, JavaScript, Python, Spring |
| Bills | bill-payment | /bill | C++, Java, JavaScript, Python |
| Bills | bill-payee | /bill/payee  /bill/payee/search | Java, JavaScript, Python |

*\*When formatting service name for a C++ service, the word ‘cplus’ is used instead. Ex, create-service-cplus*

### 3.2.3 Host Gateway

The host gateway is part of the bottom most layer to Skydots layered architecture. It fronts backend legacy services and databases. These legacy endpoints could return data in any kind of format. For this project there are two return types: JSON and XML (WSDL).

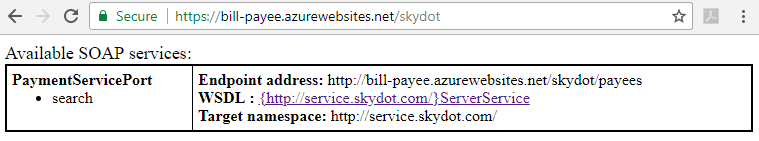


There are two parts to the host gateway: the SQL part and the WSDL part. The bank database holds all bank account data, this includes profile data, account data and transaction history. The payee service is a SOAP service that provides a list of bill payees that the user can make bill payments to. The payee service represents legacy systems that use older technologies like, in this case, WSDL. The bank database utilizes a SQLServer and represents newer technologies. The host gateway transforms data returned from both into a format suitable to return to the micro-services. For this project, JSON is the data type used within the cluster, so the host gateway converts the data from the payee service, XML, to JSON and just acts as a passthrough for the returned data from the SQLServer. This is no business logic done within the host gateway.

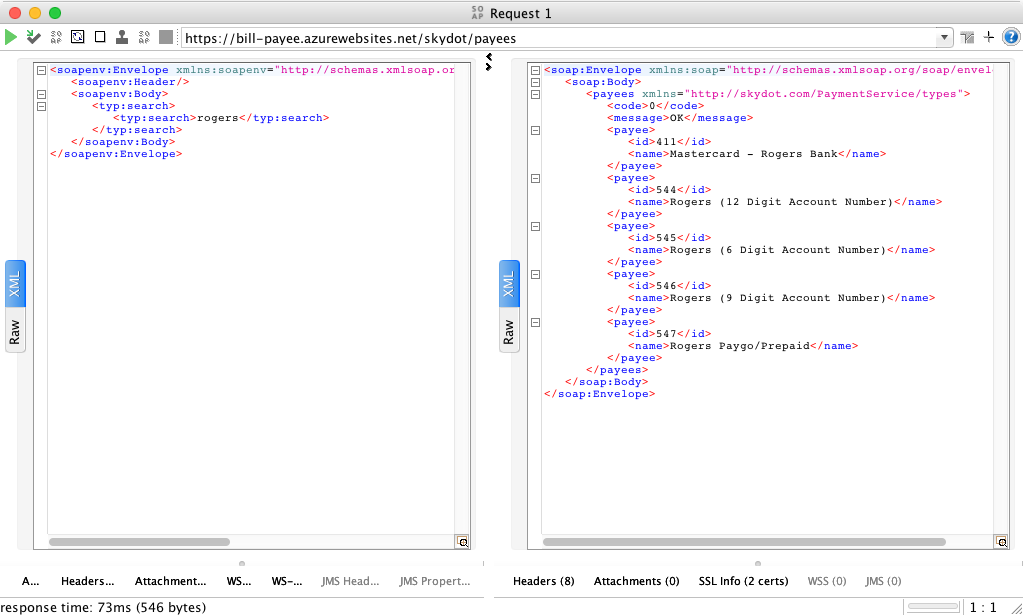
In the original design, the host gateway would be located within the cluster so that Kubernetes could provide scaling for it. Unfortunately, due to the network I am testing on, outbound requests on port 1433 are blacklisted which is the port SQLServer runs on. Without access to port 1433 I cannot test Skydot locally. The work around for this is to have the host gateway running outside of the cluster. Due to this, there may be a slight increase in latency.

## 3.3 Backend

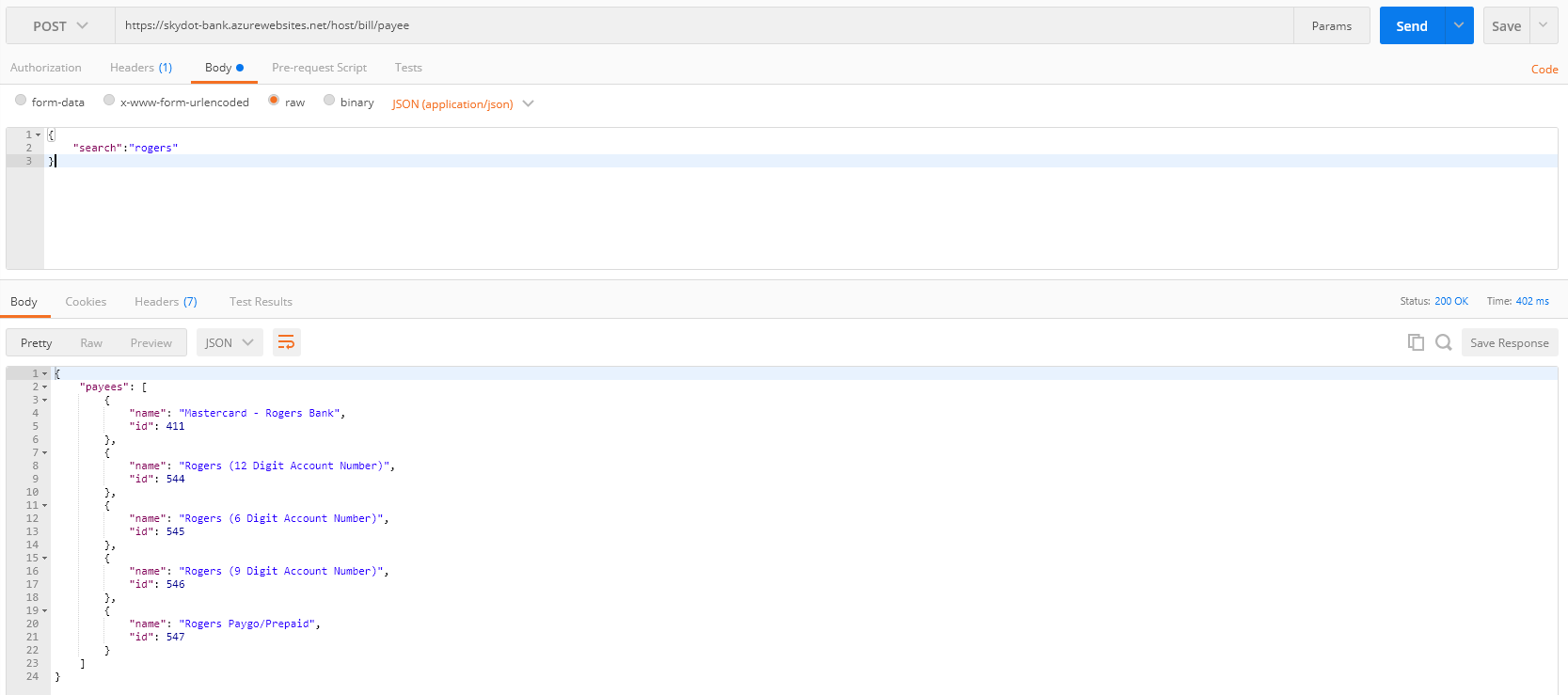
As mentioned above in the host gateway section, there are two parts to the backend. There is a SQL database that contains all bank related information and a payee service that returns a list of available bill payees. The SQLServer represents technologies currently used in industry while the payee service is a SOAP service and represents legacy technologies. These legacy systems utilize out-of-date technologies but are usually too big or too expensive to upgrade or convert to a newer solution. This flaw makes it so that the legacy system must be used instead of implementing something new. The payee SOAP service is available at the following endpoint:



And preforming a search on the service returns a response like this:



This is the type of response (on the right) that is parsed by the host gateway into JSON data. The following shows the host gateway handling the same search request but now JSON is returned:

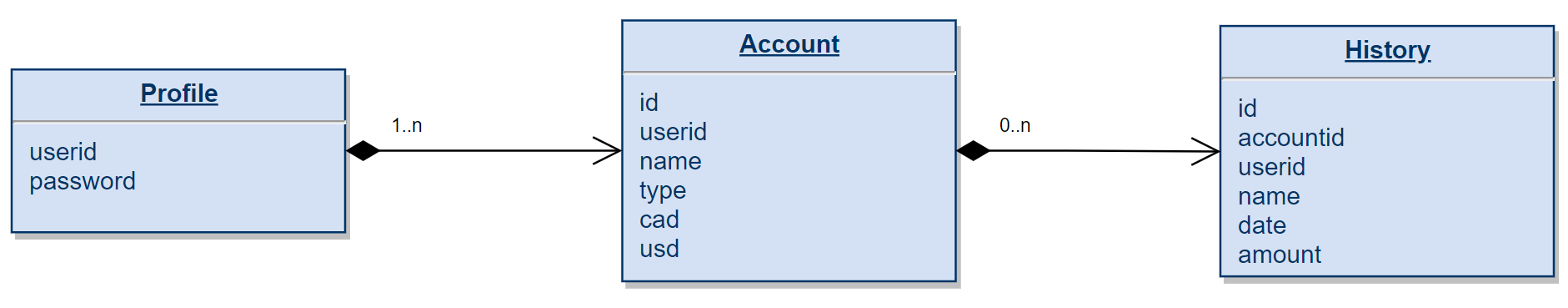


As for the banking SQLServer, there are three data tables that hold banking information. The first is the Profile table. This table holds a users’ user id and password which are used to verify a user when they log in. The user id is used in all three tables to connect each account and transaction to the appropriate user.

The second table is the Account table. This table contains data pertaining to each account a user has. Each account has a unique id, a name, a type and an amount in both Canadian and American currency. There are three types of bank accounts: banking (e.g., debit), borrowing (e.g., credit) and investing (i.e., investments but not brokerage). With different types of accounts business rules can be enforced by the micro-services. Rules like investing account can’t transfer to other accounts or banking accounts can’t pay more than its available funds.

And lastly, the third table is the History table. This table contains transaction history for each account. When a transfer or bill payment is made, a record to make when all information relevant to the transaction (i.e., date, amount, account id).

The entity relationship of the database is as follows. A profile must have at least one account, but an account doesn’t have to have any transactions.

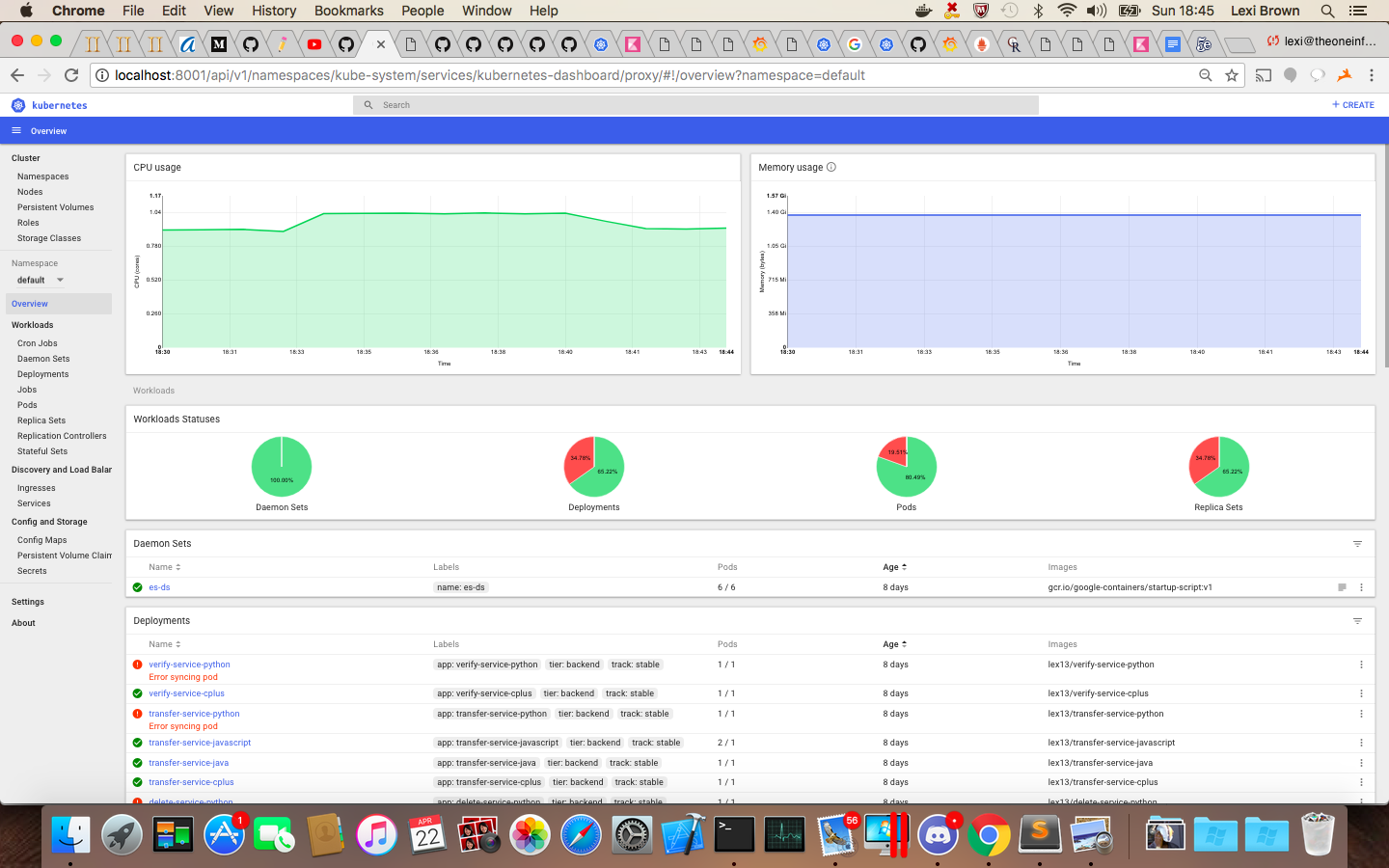


## 3.4 DevOps

### 3.4.1 Dashboard

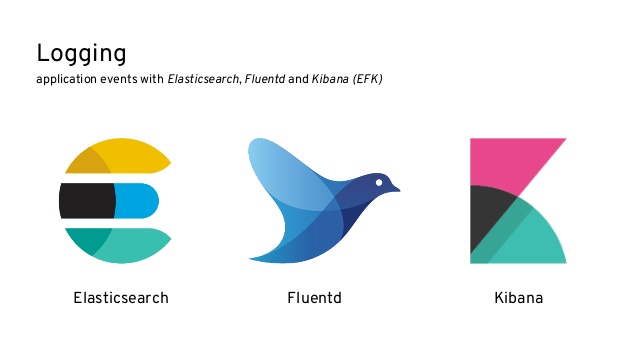
Kubernetes offers a web-based user interface that delivers an overview of the cluster Skydot is running on. However, not only does the dashboard give an overview of the applications running on the cluster, it also provides the ability to deploy containerized applications, troubleshoot containerized applications and manage cluster resources. The dashboard can be used to create, or modify, all kinds of resources within Kubernetes such as Deployments, Pods, DaemonSets, Services and much more. It also provides an API for scaling, updating and restarting pods. Logs and YAML definitions can be viewed in the dashboard and the Kubernetes dashboard even provides CPU and Memory usage graphs for each pod.

An alternative dashboard was not needed as the out-of-the-box dashboard was a suitable for this project.



### 3.4.2 Logging

The logging that was used is EFK style logging. EFK logging is the usage of Elasticsearch, Fluentd and Kibana to organize and display logs. Within Kubernetes, each pod logs information in its own way, some just stream to stdout while others generate log files. Fluentd is used to find each pods logs and parse, filter and enrich them. It then allows Elasticsearch to take those compiled logs and index and store them for quick and reliable searching. From there, Kibana uses Elasticsearch as a data source for its web-based dashboard. The dashboard provides searching capabilities, through Elasticsearch, and graphical representations of the logs.



An alternative to EFK is ELK, which utilizes Logstash instead of Fluentd. However, there isn’t much of a difference between the two as Fluentd was inspired by Logstash

### 3.4.3 Monitoring

There were two monitoring system applied to Skydot: InfluxDB and Prometheus. Both provide monitoring for Kubernetes pods and nodes, and both utilize Grafana. Grafana has a web-based dashboard that offers many kinds of graphs to show metrics.

InfluxDB was used first as it gives basic metrics readings and then Prometheus was added on later as it gives more detailed information on metrics. This is because InfluxDB is a push-based system whereas Prometheus is a pull-based system. A push-based system requires the application tracking metrics, in this case Heapster, to actively push data into the monitoring system. While a pull-based based system fetches the metrics values from Heapster periodically. The centralized control of how polling is done in Prometheus makes it easier to adjust configurations and can act as a synthetic health check monitor. Although Prometheus delivered more out of the box, both were utilized, and all metrics collected in this report were either monitored through InfluxDB or Prometheus.

# 4 Results/Validation

TODO

## Project Goals

1. Decrease the number of people needed to maintain software and the cost of maintaining that software.
2. Increase code integrity and decrease code development and integration time between teams.
3. Counter long, multi-step manual deployment with simple, quick, autonomous cloud deployment.
4. Handle cloud resources as efficiently as possible.
5. Provide a common point of access for client applications.
6. Create a layered, microservice framework that separate client applications from common services.
7. Provide a common point of access to host services and data.
8. Utilize Skydot as the server side of a mobile banking application to present the capabilities of the project.

# 5 Conclusion

TODO - tie into results and validation

In conclusion, the goal of Skydot is not just to use a “Micro Architecture” within the cloud. It’s a culture and an end-to-end process. Using all technologies and designs I’ve chosen for Skydot, all project goals can be met. However, even with these decisions, Skydot is decoupled enough that integrating a new technology (i.e., adding Cassandra) or shifting to a different technology (i.e., Kubernetes to Docker Swarm) would not bring down the whole system and require extensive conversion time. Skydot is a template that any company will be able to use, understand and customize for their needs. When used properly, one should be able to see the improvements Skydot brings to their application within cost, maintainability, developer management, productivity and adaptability/flexibility.

## 5.1 Future Work

After implementing Skydot within Microsoft Azure, future considerations would be integrating a multi-cloud design. With CICD and the flexibly of Skydot, it would be possible to utilized both Microsoft Azure and Amazon Web Services (AWS) (Figure 11 - Possible future Skydot usage with AWS).

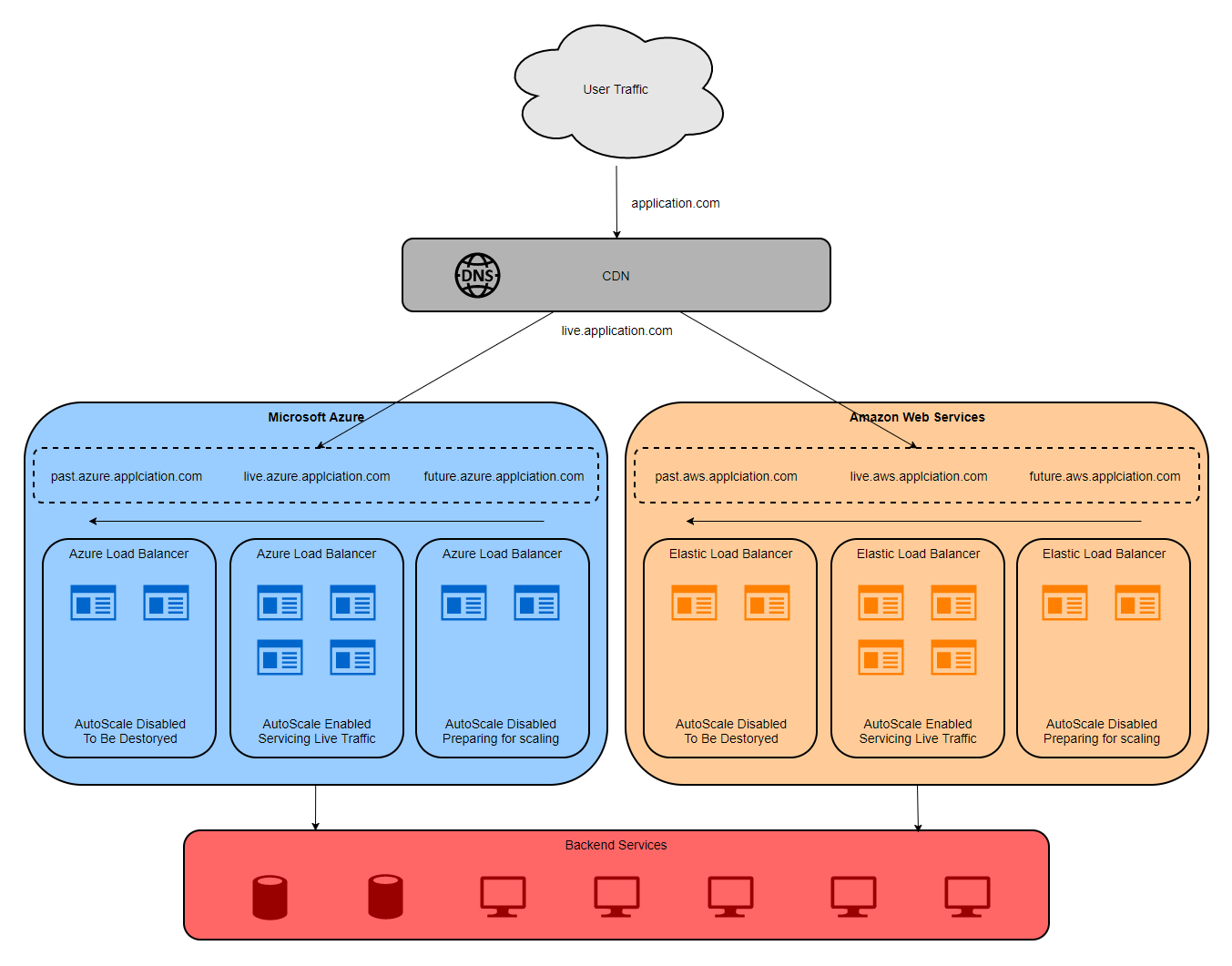
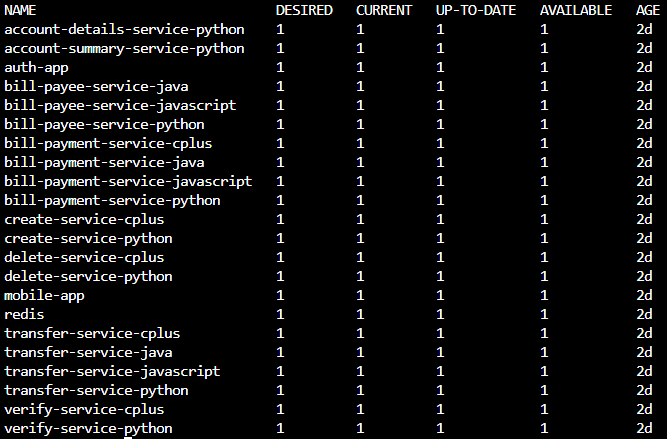


Figure - Possible future Skydot usage with AWS

This way I could compare the cost and usability of Azure and AWS, and show off Skydots portability, while also presenting a possible multi-cloud implementation. With an added layer between the client applications and the cloud services, a content delivery network (CDN) could be added to handle the load balancing between Azure and AWS. Additions would be a DNS cycle on both cloud services where the CDN looks for the live DNS on the cloud hosts to direct traffic to and the cloud hosts also have a past and future version of Skydot behind different DNSs. The past and future DNSs would help compare and test old and new versions of micro-app and micro-services in both environments as it is possible Azure and AWS could handle Skydot differently.

1. DevOps: Dashboard, health checking, logging, monitoring, continuous integration and continuous development (CICD).
2. Populate a string database with error and warning messages (en\_CA and fr\_CA).
3. Build an iPhone and/or tablet application.
4. Integrate oAuth2 and LDAP authentication.
5. Use Swagger for design and documentation.

# Additional Figures



# References