|  |
| --- |
|  |
| Student Job Engine |
| An API platform for location-based jobs |

|  |
| --- |
| Author: Janith Perera  Supervisor: Veronika Szucs  5-3-2019 |

Table of Contents

[Introduction 2](#_Toc8057066)

[Background of the Problem 3](#_Toc8057067)

[PaaS (Platform as a Service) 4](#_Toc8057068)

[Platform Architecture 4](#_Toc8057069)

[Rest API 5](#_Toc8057070)

[Authentication JWT 6](#_Toc8057071)

[JSON Web Token structure 6](#_Toc8057072)

[User Authentication 7](#_Toc8057073)

[Mux Auth Implementation 8](#_Toc8057074)

[Object Relational Mapping 9](#_Toc8057075)

[Use Case Diagram 10](#_Toc8057076)

[Search Jobs 11](#_Toc8057077)

[Haversine formula 11](#_Toc8057078)

[SQL Implementation 12](#_Toc8057079)

[Packaging and Installation 13](#_Toc8057080)

[Packaging 13](#_Toc8057081)

[Installation 13](#_Toc8057082)

[Integration 14](#_Toc8057083)

[Login 14](#_Toc8057084)

[Home 15](#_Toc8057085)

[Conclusion 16](#_Toc8057086)

[Bibliography 16](#_Toc8057087)

# Introduction

Student Job Engine is an API platform for any web or mobile interface which can provide a user interface and service to the user. The concept behind the platform is, with the growing number of exchange students from both international and local, the demand for student jobs get increased. As it is not a main objective for the university to facilitate part time jobs or temporary jobs for students, the problem of finding a job will get much more difficult for the student. The other problem is that the alien nature of the city, country and the society will make the problem more complicated. However, the student jobs in particular area play a huge role in student’s life, because most of the students require some financial support by their own. Not only the financial support, but also to adapt the environment, it is necessary to work with the surrounding society.

Some of the common methods of finding a job are, by newspaper, websites, university notices, course web (not in every case), and also by meetups. However, most convenient method would be finding a job via website or any related online material. However, the major problem with such a method is that, the student mostly likely will have no idea about the location, distance and time it takes to go there. The reason for this problem is, mostly these job portals are targeting the type of the job than the location.

The Student Job Engine platform is a prototype project, which propose a location-based job search and also job store for third party web or mobile portals. It uses google map services and RESTful API to act as a web service. The project is written in Golang programing language, with support of many Golang libraries and it runs as a backend service, which enables a port for the remote extensions to connect and communicate. It also includes an Object-relational Mapping platform to use variety of database types as some databases provides different features compare to another. This makes the platform more flexible in migrating to a new database system, and requires minimum alteration to the program.

The correct implementation of the project, with aid of a cloud platforms, proper database system and secure networking methods, can bring an impressive platform for multiple interfaces to serve many users concurrently. Other implementations can be also achieved by this project with minimum alteration to the structure of the platform and code, example, renting apartments, rooms, houses and also selling cars. The entire code base is released under open source license, where external contributors can join and improve the code, bring more features, enhance security and also change the targeted category. Main purpose of the making the source code available to the public is to collaborate with other ideas and bring a much more advanced platform for anyone who’s willing to use it. Even though it’s an open platform, further development can be led to in business perspective, with marketing strategies.

As it mentioned earlier, this is a prototype to bring a concept alive and it has many core implantations that require technical walk through to understand.

# Background of the Problem

The most common way of searching for a Job is by googling or going to a famous Job web site. These kinds of methods are popular among the normal people, who live in a city, hence they know about what they are searching, where the location, etc. However, from the student perspective, the situation can be different. Let’s consider a student (Bob) comes to a city called Shire. His main purpose of being in the city is to study his bachelor’s degree in a famous university located in Shire. Upon arrival, he does his registration and residency process and starts his regular studies by visiting the university. In most of the cases students choose their hostel or room closer to the university. In this scenario, the public transport is not very convenient in the city and the university also not located in the center. After a few months, Bob finds the expenses of that city is much higher than he estimated before he arrives and, he wants to engage with the society outside the university environment. So, he decides to find a part time job, as he’s an active student. With the lack of experience in the city and the environment, he finds it’s difficult to land a job and doesn’t find much resources to search on.

Not only there are no enough materials to search on, but also, he has new problems, they are that the location of the job, and how long it takes. As he’s continuing studies, he wants to manage the part time job and spend less time on the road for travel in between. The city itself has no convenient public transport and also private transport is unaffordable for a student in regular basis. By considering Bob’s situation, the new requirements arise in search of jobs, is that the search being location oriented. In most of the cases, part time jobs for students are very odd, and not in a specific industry or doesn’t require much skill set. Mostly they are training oriented and also contract basis. Because of these behaviors of the requirement, the targeted area would be the location, which should get high consideration from the student perspective.

The solution must be, a search performed with the coordinates of the currently location. Then based on those coordinates, with a special algorithm, the search should be performed, and result must be displayed in the user interface. The most job portals are not designed in this way, that they require exact current location of the user, rather they prefer a city. Other problem with popular job portals is they mainly target on more professional work, which are either fulltime, internships or long-term contract part time jobs. These are mainly provided by the companies, where they expect much documentation process to hire new employees and they always consider a fixed payment, and much more other conditions. For the students like Bob, it is very convenient, if the University provides such a job search feature in any of their portal, where students require no other 3rd party website to depend on. It will make more trust worthy service for the students and, they can get some help from University officials for further information. Although it sounds very soft implementation, the University might not be able to handle everything by their own, where they require an open platform to use as a service.

# PaaS (Platform as a Service)

Student Job Engine is completely a service-oriented platform, which runs as a backend service. In this section it talks about the main architecture of the platform, database layer and RESTful API layer. The platform is purely written in Go language, which is a statically typed, compiled programming language designed by Google. It is syntactically like C, but it includes memory safe, garbage collection, and supports for structural typing, which are not available in C, unless they are implemented within the project. As it is compiled programming language, the use of resources is minimum, and the speed is higher than other competitive programing languages like Python and Java.

## Platform Architecture

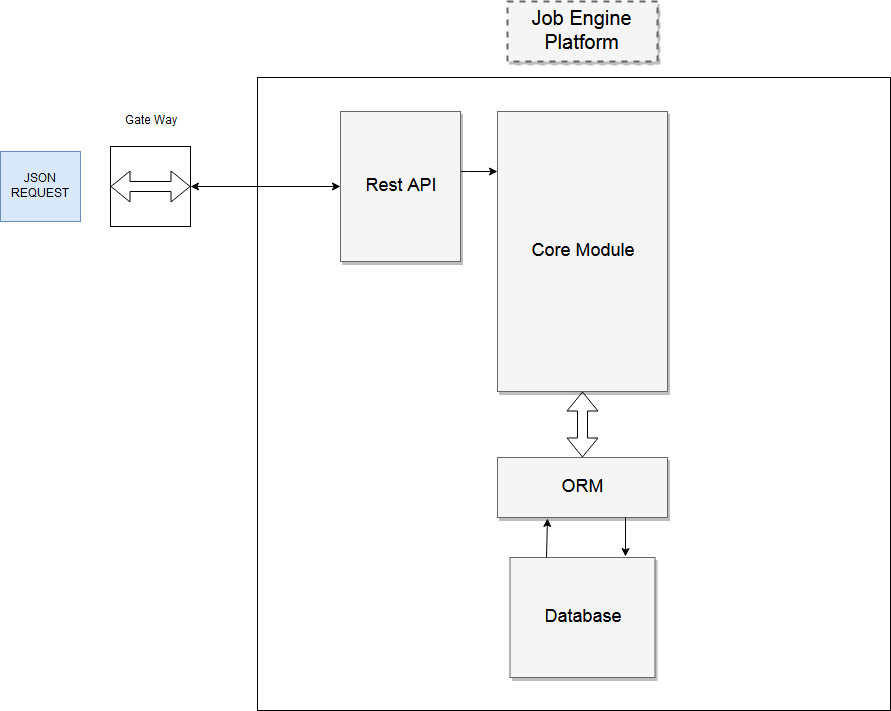


Figure 1Platform Architecture

# Rest API

This is the so-called front end of the platform, where the third-party applications can interact with the platform. It should be connected with a properly secured gateway to handle requests from outside. The Rest API is implemented with Gorilla Mux package, which is designed for Go environments. Gorilla Mux implements a request router and dispatcher for related incoming requests via web interface. The Mux stands for “HTTP request multiplexer”.

The implementation of Gorilla Mux is available as a part of core module, even though it is linked to the Rest API. The **main()** method initiates a Gorilla Mux router, and then handle the requests based on the path, and the method. The following implementation shows how it linked with user operations. Furthermore, in this implementation, it uses two Muxes for authentication required and not required operations. In later section, the authentication will be covered in detail.

    router\_noauth := mux.NewRouter()

    router\_auth := mux.NewRouter()

    router\_auth.HandleFunc("/jobs/{id}", ops.GetJobById).Methods("GET")

    router\_auth.HandleFunc("/jobs/add", ops.InsertJob).Methods("POST")

In the example, it shows two methods, one is to get all users and the other is to add a user. With a close consideration, it can be seen that, the methods required to access these registered paths are **GET** and **POST** respectively. Also, the two **mux** routers are dedicated for both authentication and non-authentication.

The REST API part of the platform consist of several subroutines for every operation that the platform has to do. Every subroutine require **http.Request** and **http.ResponseWriter** objects as arguments, where it can gather http request parameters and write back as a response. The entire reading, operation and writing back to the client is handled by within the subroutine.

The following code snippet shows an example of subroutine implementation. Each of these subroutines will be called in the **HandleFunc** of **Mux** router, with appropriate method.

func GetUserByUserName(w http.ResponseWriter, r \*http.Request){

    db := orm.Connect\_To\_Database()

    params := mux.Vars(r)

    user\_name := params["user\_name"]

    var user orm.User

    user = orm.Search\_User\_From\_User\_Name(user\_name, db)

    json.NewEncoder(w).Encode(user)

}

## Authentication JWT

JWT (JSON Web Token) is an open standard for securely transmitting information between parties as JSON object. Hence these are digitally signed, they can be verified and trusted. For the signing process, it can be either secret or a public/private key pair, which uses **RSA** or **ECDSA**. Although JWT is used in purpose of securing the transmission, it is not work as password authentication. It is mainly used to sign tokens. Let’s consider some of the practical situations where JWT can be applied.

* Authorization

In this project, JWT is used in authorization purpose. Once the user logged in, each subsequent request will include the JWT, which allows user to access routes of the **Mux**, which was explained in the previous section.

* Information Exchange  
    
  It is also a good method to securely transmitting information between parties. JWT can be signed using public/private key pairs, which means, they can be used to make an encrypted transmission to the sender. It is also possible to know that the content hasn’t been tempered, because the signature is calculated using the header and the payload.

### JSON Web Token structure

The compact form of the token consists of three parts, which are separated by dots.

* Header
* Payload
* Signature

A JWT typically looks like the following

**xxxxx.yyyyy.zzzzz**

* Header  
    
  The header has two parts, the type of the token, which is JWT and the signing algorithm, which can be either HMAC, SHA256 or RSA.
* Payload  
    
  This part contains the claims. Claims are the statements about the entity, and additional data.
* Signature  
    
  The signature is used to verify the message wasn’t changed while it was transmitting to the destination. It can also verify that the sender of the JWT.

## User Authentication

The user authentication is done by normal user credentials procedure, which is acquiring username and password as the login request. The username and password are set in the registration process, which will be explained in a later section. This is an example **JSON** string, that can be used to login to the API. This process also known as **signing** process.

{

"UserName":"jantwisted",

"Password":"321321212134"

}

The expected method from the **Mux** is a POST request. It is possible to test this by any tool which supports different HTML operations, however for easy documentation, in the following example it uses a GNU/Linux utility called **curl**.

curl -H "Content-Type: application/json" --data @user\_login.json 127.0.0.1:8080/users/login

In this example, it sets its header to **JSON** type and passes the JSON string from a file. Here the API is hosted in the same server, however, in the production environments, the loopback address and the port can be changed, or the port may be omitted.

As a response from the API platform, it will return the **JWT** token back to the requester. In the above example, the token will be printed to the screen. Only in this example, the format can be hard to read, as there will be no proper formatting. However the command output can be piped with python -m json.tool to make a pretty print.

curl -H "Content-Type: application/json" --data @user\_login.json 127.0.0.1:8080/users/login |python -m json.tool

Once the user gets the **JWT** token, then for every other request made to the API platform must be authenticated with the valid token. The token has an expiry date, so until the time reach that threshold, the token can be used as a valid token.

curl -H "Authorization: Bearer $TOKEN" <url>

In this command, it uses the token from an environment variable, however, it is possible to mention the token without a variable, which can be lengthy. The targeted URL can be any path, which require authentication. It is also possible to request normal URLs, where the authentication token will be neglected.

## Mux Auth Implementation

Previously, it mentioned about the exact application of the **mux** router and in this section, it talks about the separation of authentication and no-authentication. This is only a proprietary design for the project, may or may not be common for every implementation.

The two routers used in this API are, **auth** and **noauth**, for both authentications required, and not required respectively. For this authentication, it uses another library called **negroni**, which is a wrapper for **Go** http requests. In **noauth** router, it is simply passed as a user handler to **negroni**. However, for **auth** router, another inter subroutine is called as middleware to check the token, whether it is valid or not.



In this implementation, some of the parent paths are completely removed from authentication, and only applied for the child paths, which are leaves of the tree. This will prevent users to getting denied request for child paths like login, which are obviously not required any authentication.

All these methods are used in authentication part, also every authentication required path can be accessed with the requests, where the JWT token is present in the header. These kinds of request are explained in the previous section.

# Object Relational Mapping

Object-relational Mapping (ORM) is a technique used in programing, which lets the programmer to manipulate data from database (or datatype), using an object-oriented paradigm. Most of the modern programing languages are rich with ORM libraries. They encapsulate the outer method calls into a form to manipulate data structures. In other words, the programmer requires write minimal SQL statements to interact with the database system, rather calls the methods which are appropriate for certain task. This will bring flexibility from the use of database system to the API platform. Example, with this approach, it is currently using PostgreSQL DBMS, however, it can also use MySQL or Oracle or any other supported rational database system with minimum amount of code change. Let’s consider some of the application of this library in the API platform.

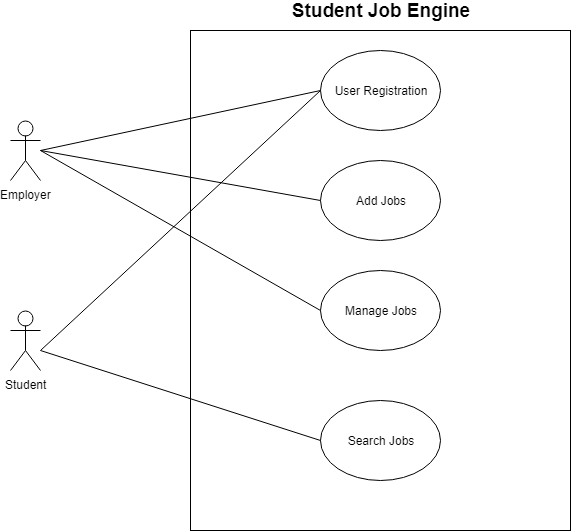
The following example shows how it adds a Job into the database. Here it uses a **Go** library called **GORM**, which is specially designed to encapsulate the database calls.



In this code snippet, it shows how to insert a job into the database. First, the database tables must be defined as **Go** data structures. Then the tables must be migrated to the database, which is usually done by a simple method call, and for this platform, this method call happens as a start up method, which means, if there’s a change in the database structure, every time it restarts, the change will be applied in the database end, which require no additional migration procedure.

Once the migration is done, the Job table will be in the database, and then in the “Insert\_To\_Job” method inserts the user, based on the data it retrieves from the given data structure, in this case it’s a JOB. The other parameter is the database instance, which must be initiated prior to the method call, usually this is very similar to classical database instance creation, by parsing details and credentials.

# Use Case Diagram



# Search Jobs

Searching Jobs based on the user’s location is a critical part of this project. The algorithm is developed on searching Jobs is based on **Haversine formula** and it is written as a SQL statement.

## Haversine formula

Haversine formula determines the shortest distance between two points on the surface of a sphere (Great -circle distance) given their longitudes and latitudes.

φ: latitude

λ: longitude

R: earth’s radius

The implementation of the search can be done in two ways, one is to get all the locations, and programmatically filter them according to Haversine formula or filter them in the database side and get only the filtered result set. Both these approaches have advantages and disadvantages. For this project, it uses the second method, which is, using the algorithm inside the query. However, with this approach, the drawback is, that the portability of ORM will be slightly loose. In other words, if the platform uses a different database system than PostgreSQL then, this SQL query must be altered. However, that also depends on the database system, as PostgreSQL mostly uses common methods.

First obstacle comes while implementing this algorithm is that how to manipulate ORM, into a custom query, as ORM has no pre-defined implementation for Haversine formula. However, the GORM encapsulation provides a nice method to include custom queries in the ORM. The following code shows how it is implemented.

var job\_list []Job

rows, err := db.Raw("CUSTOM SQL", distance).Rows()

## SQL Implementation

SELECT id, ( 3959 \*

acos( cos( radians(LAT) ) \*

cos( radians( cast(latitude as float) ) ) \*

cos( radians( cast(longtitude as float) ) -

radians(LONG) ) + sin( radians(LAT) ) \*

sin( radians( cast(latitude as float) ) ) ) ) AS distance

FROM jobs

In this SQL statement, it uses **LAT** and **LONG** parameters, which are the current location of the user, respectively latitude and longitude. The syntax is PostgreSQL supported, however, for other database systems, it will be similar. Here the radius of earth is given in miles.

The inner SQL compiles Haversine formula. The Job table also like User table, it will construct as a **Go** data structure and migrate through the method available in ORM library. When adding a Job, the employer must include the coordinates, latitude and longitude in the JSON file and it will be added into separate columns. From the inner query, it fetches all the coordinates available in the table and then do the calculation according to the mathematical algorithm.

SELECT X.id FROM (

SELECT id, ( 3959 \*

acos( cos( radians(LAT) ) \*

cos( radians( cast(latitude as float) ) ) \*

cos( radians( cast(longtitude as float) ) -

radians(LONG) ) + sin( radians(LAT) ) \*

sin( radians( cast(latitude as float) ) ) ) ) AS distance

FROM jobs ) X

WHERE X.distance < DIS ORDER BY X.distance

With the combination of outer SQL, which is a wrapper to the Inner SQL, sorts the jobs based on the returned distance value from inner SQL. From this approach, it will give the result, where the distance is less than the **DIS** parameter, which is passed to the SQL statement.

The DIS parameter must be an integer as there’s no cast operation implemented. LAT and LONG parameters should be strings, where they will be casted to floats before constructing the mathematical algorithm.

However, all parameters and SQLs are defined within the platform ORM custom package, there’s no dependency on the format of request, only they must be valid coordinates in order to process without a problem.

# Packaging and Installation

## Packaging

The packaging is an additional but still important task, as the platform has built to run in a server environment. For the packaging purpose, it is also possible to follow a manual directory structure and write a **Makefile**, however, to follow a standard, it uses cookie cutter third party opensource project. Cookie cutter is originally designed to package Python projects; however, they have templates for other languages like **Go**.

## 

## Installation

Before install the platform, make sure to setup **Go 1.11** and **glide** package manager (The package manager is necessary, only if the user decide to install all the dependencies automatically). When setting up the **Go** environment, make sure to setup all the necessary environment variables.

* Clone the project or download the zip file from the repository.  
    
  git clone <https://github.com/jantwisted/studentjobengine.git>
* Install dependencies, this will read glide.lock and glide.yaml files.  
    
  glide install
* Enter to the directory, clean the binaries (can be skipped) and build  
    
  make clean && make build
* Setup environment variables which are listed in the README.md file. Example  
    
  export DBTYPE=Postgres

Export DBHOST=127.0.0.1

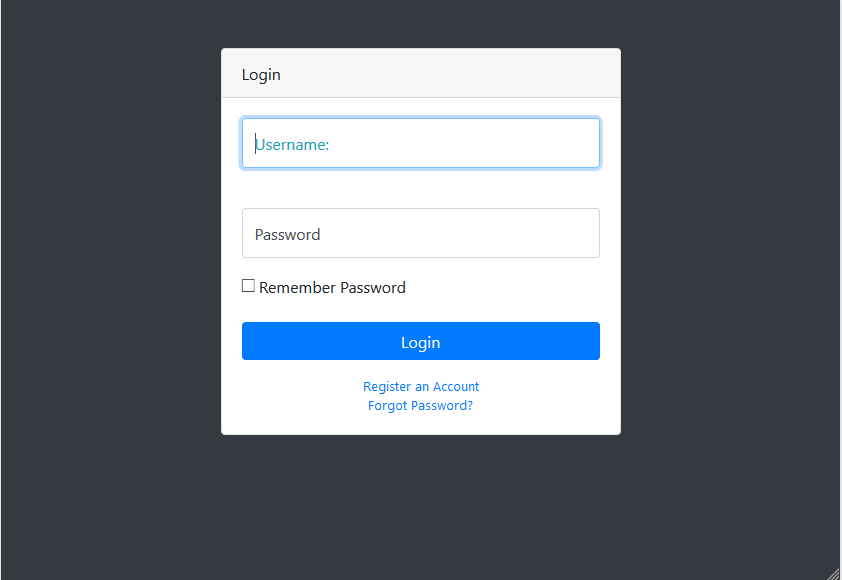
DBPORT=5432

* Once everything is done, execute the binary file available in the **bin** directory.
* Note that the environment variables are mandatory to run the platform.

# Integration

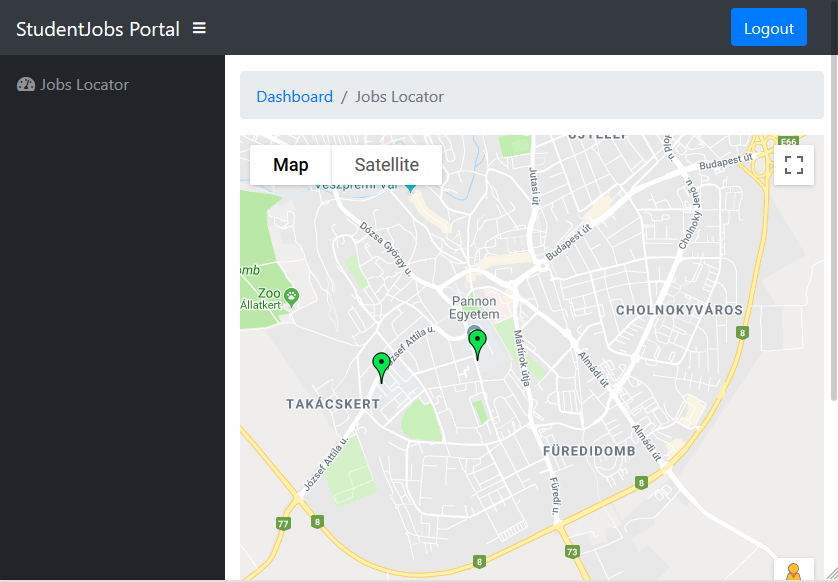
In this section it shows some interfaces from the user application. For this example, it uses a web interface.

## Login



From this form, it sends user credentials to /users/login, and retrieve JWT token as the response and save it in the memory for further use.

## Home



This is a sample home page, where it shows a map with marked two Jobs. Once the user login, then it will automatically send a GET request to /jobs/search path.

In this implementation, it uses Google Maps API, with API key, stored in an environment variable. The Google Maps API has inbuild features to get location marked based on the provided array of coordinates.

As it mentioned before, same as this example website, the 3rd party interface can be anything. A desktop application, a website or a mobile app. The recommended way is to use a mobile app, because it is easy to get the current location coordinates of the user. In other methods, there are some alternative ways of getting the current location, however none of them are straight forward.

# Conclusion

The Student Job Engine platform is a result of innovative implementation, use of programing concepts, communication and planning of distributed architecture. It provides flexibility to the 3rd party developers who wants to use the platform as a service. They do not have to write something from the scratch, instead simple API calls can be made. This will help them to focus on their business model and implementation, then the functionality of the core module. The platform is open for custom functions, structures and different categories. Not only the flexibility of the functionality, but also the proper packaging and use of layered architecture to build one module on top of the other makes the ability to deploy it anywhere, such as containers, physical servers, cloud platforms and serverless platforms.

It can be concluded that this project provides a better solution for the problem, where students face finding an appropriate distance-oriented job, than the popular job search engines which are available in most of the places.

# Bibliography

|  |  |
| --- | --- |
| [1] | L. Richardson, M. Amundsen and S. Ruby, RESTful Web APIs: Services for a Changing World, O'Reilly. |
| [2] | "Introduction to JSON Web Tokens," [Online]. Available: https://jwt.io/introduction/. |
| [3] | K. Roebuck, Object-Relational Mapping (Orm), Samford, Australia: Tebbo, 2011. |
| [4] | "The Go Programming Language," [Online]. Available: https://golang.org/doc/. |
| [5] | G. A. Korn and T. M. Korn, Mathematical Handbook for Scientists and Engineers, New York, USA: Dover Publications, Inc.. |