

|  |  |  |
| --- | --- | --- |
| **MODULE** | **:** | CT071-3-5-3-DDAC - DESIGNING & DEVELOPING CLOUD APPLICATIONS |
| **STUDENT NAME** | **:** | HUA CHYE YEE |
| **STUDENT ID** | **:** | TP032037 |
| **INTAKE CODE** | **:** | UC3F1701SE |
| **LECTURER** | **:** | DR. KALAI ANAND RATNAM |
| **HAND OUT DATE** | **:** | 20TH JULY 2017 |
| **HAND IN DATE** | **:** | 9TH OCTOBER 2017 |
| **COURSEWORK TITLE** | **:** | ONLINE FLIGHT BOOKING SYSTEM |

# Acknowledgement

First and foremost, I would like to extend my gratitude to my parents, Lim Kin Hwa and Hua Kia Heng whom have been giving me much needed support and motivation throughout my toughest time when pursuing knowledge in Asia Pacific University. Their words of encouragement and blessings made the unbearable bearable and helped me to go through the challenging times.

Next I’d want to thank my lecturer, Dr. Kalai Anand Ratnam for pointing me in the right direction whenever I am feeling lost. His teaching and guidance are what made this project possible. Not to mention the much helpful classmates who actually spent their time to input their ideas and comments regarding the design of this project, which helped to shape a lot of areas of the interface design.

Table of Contents

[Acknowledgement 1](#_Toc494719713)

[Table of Contents 2](#_Toc494719714)

[1 Introduction 4](#_Toc494719715)

[1.1 Objectives 4](#_Toc494719716)

[1.2 Scope 5](#_Toc494719717)

[1.3 Requirements 5](#_Toc494719718)

[1.3.1 Functional 5](#_Toc494719719)

[1.3.2 Non-functional 5](#_Toc494719720)

[1.4 Deliverables 5](#_Toc494719721)

[2 Project Plan 6](#_Toc494719722)

[3 Design 8](#_Toc494719723)

[3.1 Cloud Design Pattern 8](#_Toc494719724)

[3.2 Architectural Diagrams 9](#_Toc494719725)

[3.3 Design Considerations 10](#_Toc494719726)

[3.4 Modeling 10](#_Toc494719727)

[3.4.1 Entity Relationship Modeling 11](#_Toc494719728)

[3.4.2 Use Case Diagram 12](#_Toc494719729)

[3.4.3 Sequence Diagram 13](#_Toc494719730)

[3.5 UI Wireframe 17](#_Toc494719731)

[3.5.1 Index page 18](#_Toc494719732)

[3.5.2 Sign In page 18](#_Toc494719733)

[3.5.3 Sign Up page 19](#_Toc494719734)

[3.5.4 Dashboard main/booking management page 19](#_Toc494719735)

[3.5.5 New Booking page 20](#_Toc494719736)

[3.5.6 Checkout page 20](#_Toc494719737)

[4 Implementation 21](#_Toc494719738)

[4.1 Application Development 21](#_Toc494719739)

[4.2 Azure Publishing 25](#_Toc494719740)

[4.3 Application Scaling 27](#_Toc494719741)

[4.4 Testing 28](#_Toc494719742)

[4.4.1 Unit Testing 28](#_Toc494719743)

[4.4.2 Performance Testing 33](#_Toc494719744)

[4.4.3 Test Analysis 35](#_Toc494719745)

[4.5 Managed Database (PaaS) 43](#_Toc494719746)

[5 Conclusion 47](#_Toc494719747)

[6 References 48](#_Toc494719748)

[7 Appendix 49](#_Toc494719749)

# Introduction

Ukraine International Airlines (UIA) is a flagship carrier and the largest airline in Ukraine. It operates domestic and international passenger flights and cargo services to Europe, the Middle East, the United States, and Asia. The airline is looking forward to expand into new markets but was held back by technical and security issues, rendering them unable to serve clients from many parts of the world.

Therefore, UIA is looking at designing and developing a new, more secured and reliable Online Flight Booking System which should offer a better browsing and booking experience to their customers while helping them to reach more customers from different parts of the world. Dmitriy Prudnikov, the Chief Information Officer at Ukraine International Airlines speculated that the aforementioned issues can be overcome by the new system, which will be deployed to the public cloud.

The airline company came to a decision to use Microsoft Azure after assessing its features and qualities with its competitor, Amazon Web Services as the former offers higher compatibility with open source software.

## Objectives

The objectives of this project are:

* To design and develop a robust online flight booking system
* To design and apply appropriate design pattern or cloud architecture to make the solution scalable
* To test and analyse the performance of the solution under heavy load

## Scope

The scope of this project is to plan, design and develop an Online Flight Booking System and deploy it on Microsoft Azure. The design and implementation will only cover the web application and database aspect of the system. Additionally, this project also covers the deployment of the web application and database as well as configuring them on Azure to ensure the meet the necessary requirements. Testing will also be performed on the web applications to measure the crucial metrics of the solutions, and an analysis will be conducted based on the acquired data from testing. Maintenance works, however, are not part of the scope of this project and thus, will not be performed.

## Requirements

The functional and non-functional requirements of the system are detailed in this section.

### Functional

* User can register as member
* Registered user can login
* Registered user can browse and book flight tickets
* Registered user can view their bookings in detail
* User can change the selected currency when they are browsing on the website

### Non-functional

* Able to serve 500 concurrent users with an average of 100ms response time
* Able to scale up to meet the increasing number of load
* Able to protect user account from brute force, rainbow and dictionary attacks

## Deliverables

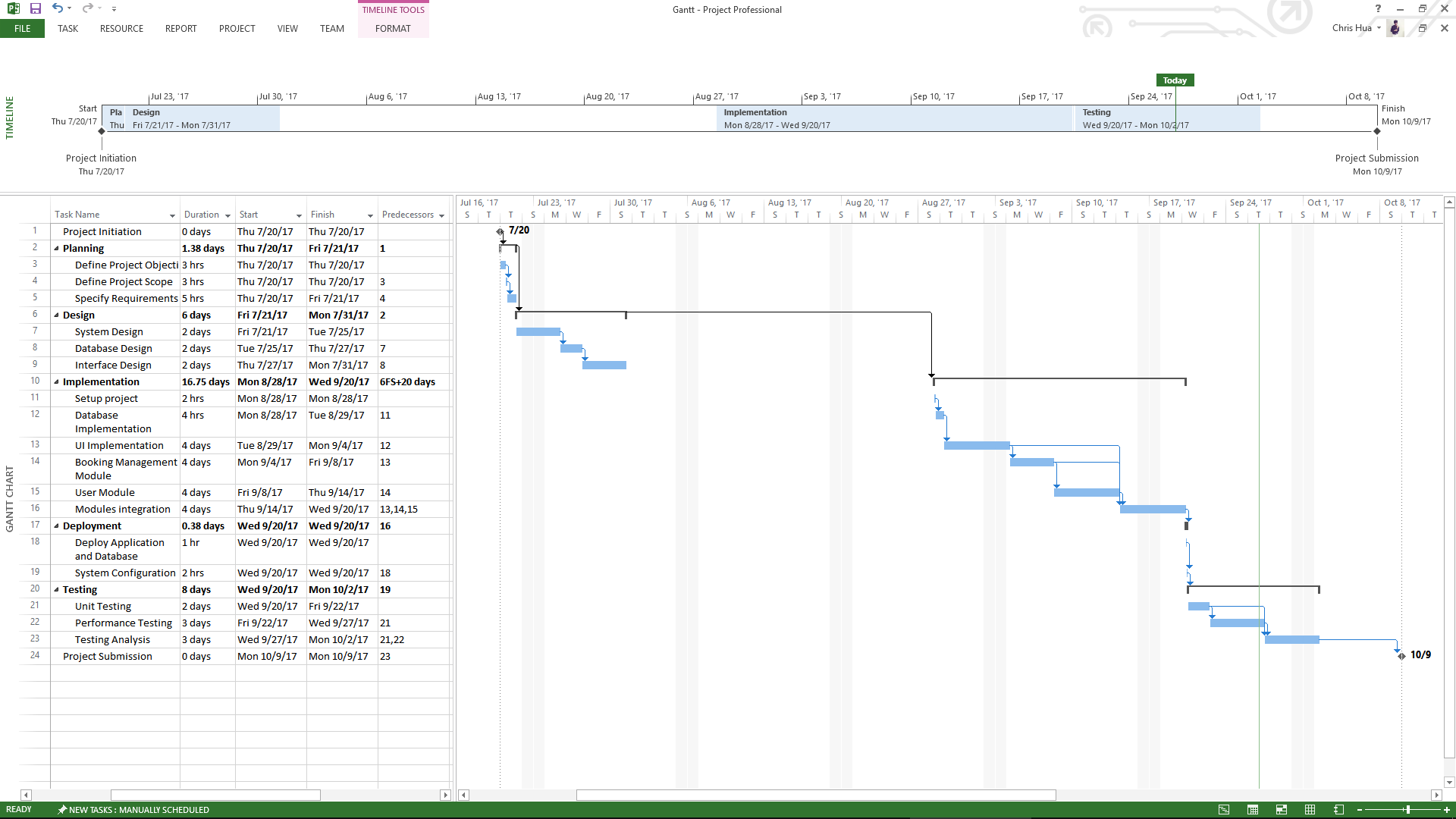
The deliverables of this project include the web application itself as well as the web application configuration file and its back-end SQL database.

# Project Plan

Before starting development on the project, its objectives, scopes and requirements will be defined in planning stage. The details will then be used to shape the functionality and design of the system. With the information acquired, the architecture, modules, database and interface of the system will then be described using UML diagrams in the design phase.

After getting the necessary plan and design out, system development will be commenced. The necessary tools, frameworks and libraries will be acquired and project repository will be created on GitHub. Then Microsoft Azure will be configured to enable continuous code deployment via GitHub. After that, the database, user interface and system modules will be implemented, developed and integrated, forming a complete system.

After the project has been fully developed, it will be tested locally before being deployed to Azure. After the deployed to the cloud, necessary configurations will be made as required such as implementing traffic manager and setting up end points as well as configuring auto-scaling. Lastly, the system will be tested again, in addition to running several performance tests on web site.

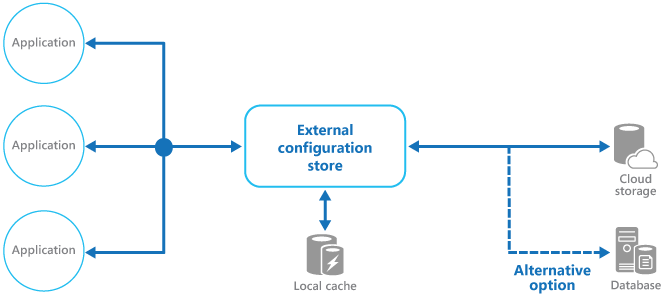


# Design

This section describes the design choices used for the web application and its database.

## Cloud Design Pattern

Since the application will be deployed in multiple regions, the external configuration store pattern can be extremely useful in this case. Such pattern stores the application configuration in an external source, in which then will be read or updated by the applications (Microsoft, 2017). By centralizing the configuration file in one location, it makes the configuration file easier to be managed and controlled. An example of the external configuration store pattern is showcased as below:



In this project, the configuration file will be stored in an external blob storage account and will be loaded whenever the applications receive HTTP request from users. The actual pattern adapted for this project can be seen in next section.

## Architectural Diagrams



The web applications will be deployed across different regions as mentioned previously. Traffic Manager will be utilized to manage and redirect the traffic to the web application which is located closest to the user location. This ensures the visitors or users can get the best performance when browsing the website. In order to ensure the data consistency across the web applications, both web applications will be sharing and accessing the same database. Additionally both web applications will also be sharing the same configuration, which is stored externally.

Both of the web applications were setup to be deployed continuously from GitHub and each web application has its own branch. Thus, all commits or changes made to the branches will be deployed automatically, making the applications easier to be maintained.

## Design Considerations

The design decisions made for the system were mostly influenced by the requirements set by Ukraine International Airlines. For instance, the web application must be scalable and capable of serving customers from various regions. Thus, the cloud architecture and design pattern selected for this system were scalable horizontally, expandable and easy to maintain in nature. Another factor that influences the design is the development budget. Since only RM150 ($25) Azure credit was given every month, the development build version of the websites will only be deployed in 2 regions. The number of auto-scaling instances and database tier will also be limited during development in order to keep the spending within the budget. The web application and its database were designed after the functional requirements, in which will allow the users to create account, login, logout, create and view booking.

## Modeling

This section will showcase the entity relationship diagram, use case diagram and sequence diagrams created for the system.

### Entity Relationship Modeling



The database design is kept as simple as possible, yet expandable in case UIA wishes to add more tables or columns to store more data. Basically, the user details, credentials and role are stored in the user table. The role is selected from user\_role table. Each user\_role will grant them access to different page, in which is stored page\_permission table. As for the page data, it is stored in page table. User session may be created and stored in user\_session table in case they opt to persist their session by checking the “Remember me” checkbox when signing in. The booking flight, route, price and airport details are stored in their self-titled tables respectively, where they will be read and write as users are browsing or booking the flights.

### Use Case Diagram



Unregistered users can use the system to register a new account, which then can be used to login and access the services on the web application. As for the registered member, they can use the system to create and view bookings. Additionally they can logout from the system, in which will clear all their sessions.

### Sequence Diagram

The sequence diagrams illustrate the processes and steps in which the user will go through when performing a specific activity or action from the use case diagram.

#### Register



#### Login



#### Create booking



#### View Booking



#### Logout



## UI Wireframe

The user interface of the system was wireframed to give an idea of how the components and web pages should appear in the final system. Do note that the wireframe shown here does not represent the final design of end product as these are merely early design mock-ups. Numerous amendments to design have been made since the first draft as additional parameters are added. Some adjustments were also made to improve the aesthetic and usability of the final system.

### Index page



### Sign In page



### Sign Up page



### Dashboard main/booking management page



### New Booking page



### Checkout page

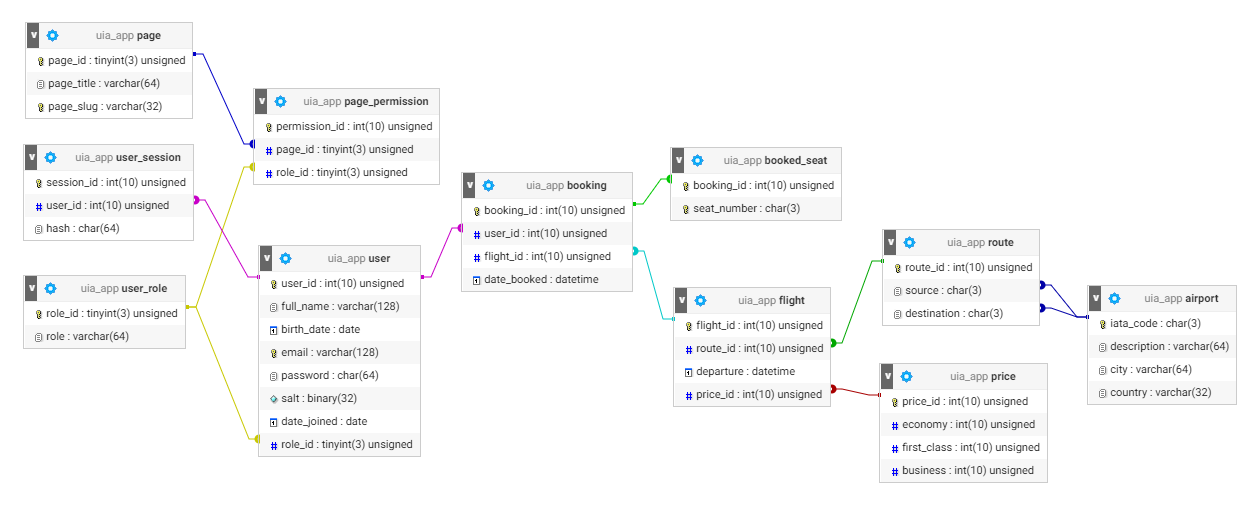


# Implementation

## Application Development

The web application was mainly developed using PHP, MySQL, HTML5, CSS3 and Javascript. Undoubtedly ASP.NET and MSSQL can also get the job done and such combo is even considered as the go-to choice when developing web app for Azure, but the former language and DBMS engine was chosen due to my familiarity with them.

The database schema was designed and created using graphical tools found in phpMyAdmin. The screenshot below shows the tables as well as their relationships created for the database.

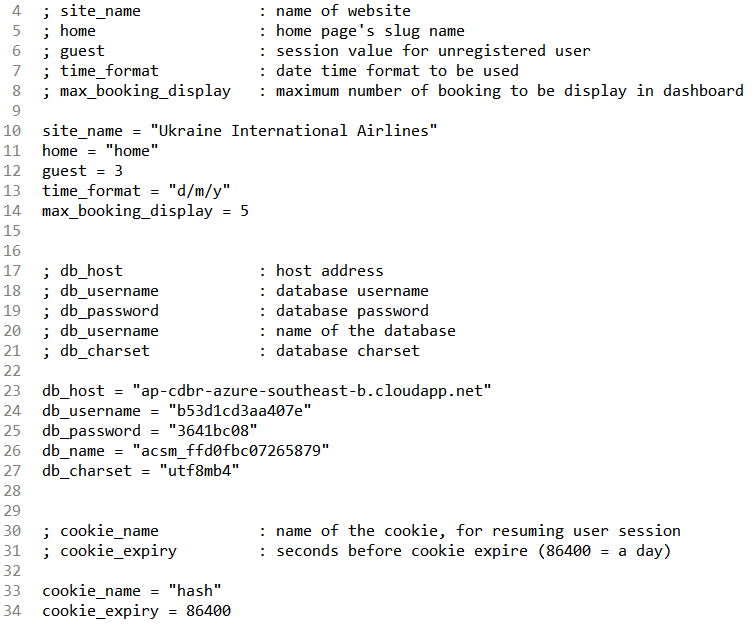


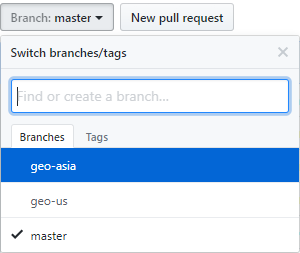


After getting the database ready, the development of the application front-end and back-end are then commenced. Since this web application is developed with Model2 MVC architecture in mind, the both front-end and back-end components were segregated. As seen from the project directory shown on the left, the “modal” directory contains the business logic and data mapper whereas the front-end components are located in “view” folder. Most “controller” is actually embedded in the respective view itself, thus it does not strictly follow the classic MVC architecture. Nevertheless, this still allow majority of the codes and components to be reused when developing the application, resulting in a shorter development besides being easier to maintain and highly extensible.

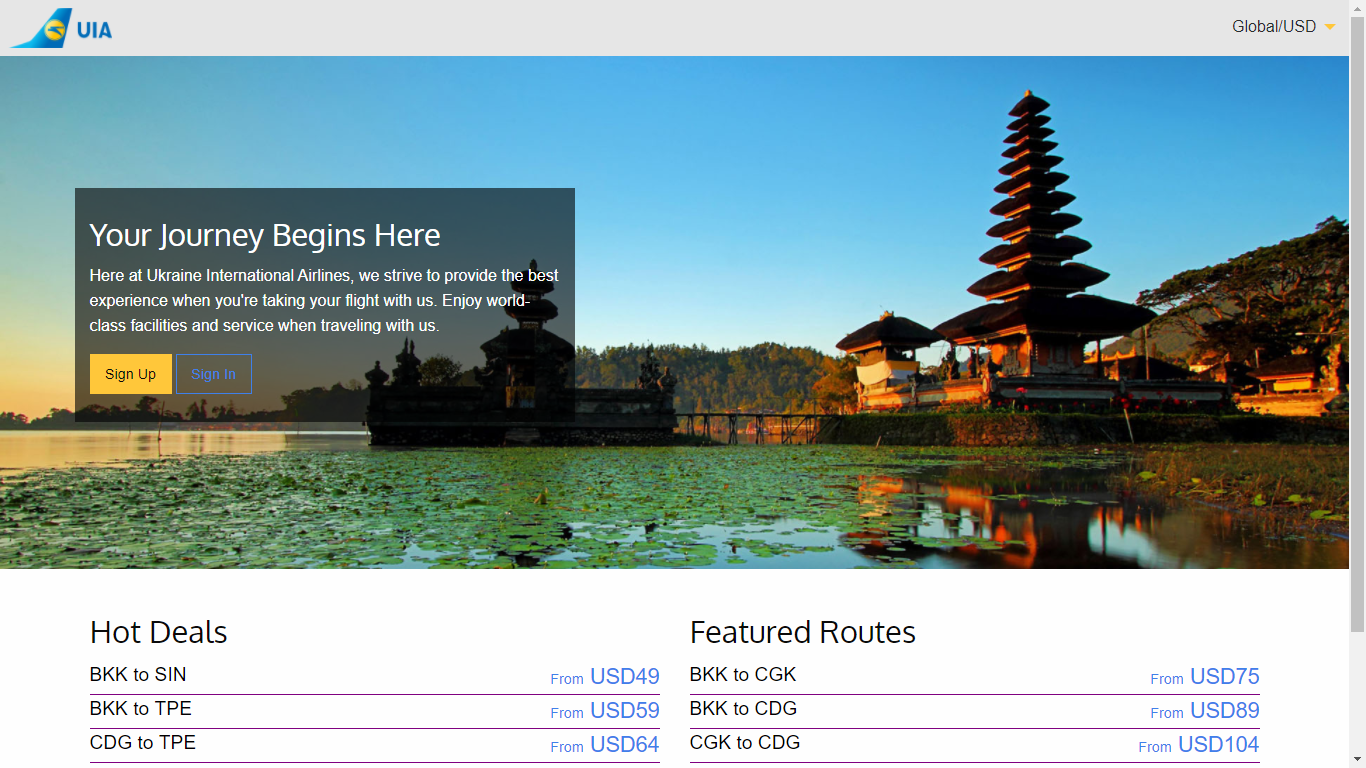
Having the front-end and back-end separated also allows them to be developed separately. During the development process, the UI components such as the menu, page layout and styles were one of the earliest things to be worked on. In order to speed up the development time, Foundation Framework and other open-source libraries were used to build the front-end of the web application. Then, the “dashboard.php”, “index.php” and “portal.php”, which are basically the template are used to load and combine front-end components from “view” folder. As for the back-end, it is consists of 2 modules, namely user module which includes account creation as well as login and booking module which covers flight booking and viewing of bookings.

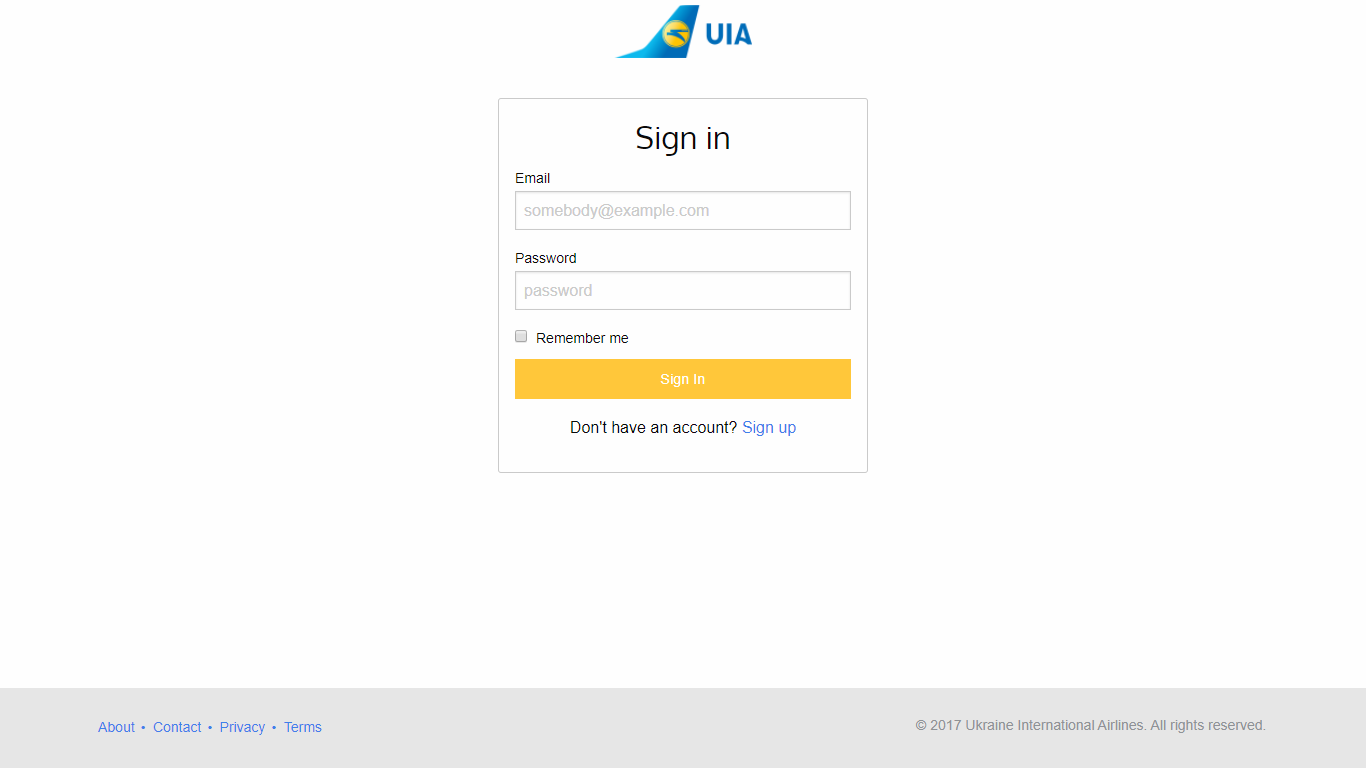
Since external configuration store pattern was adopted, the web application-wide configuration file is created and stored externally. The configuration file contains vital settings such as database connection and session settings which are required for the web application to function properly.

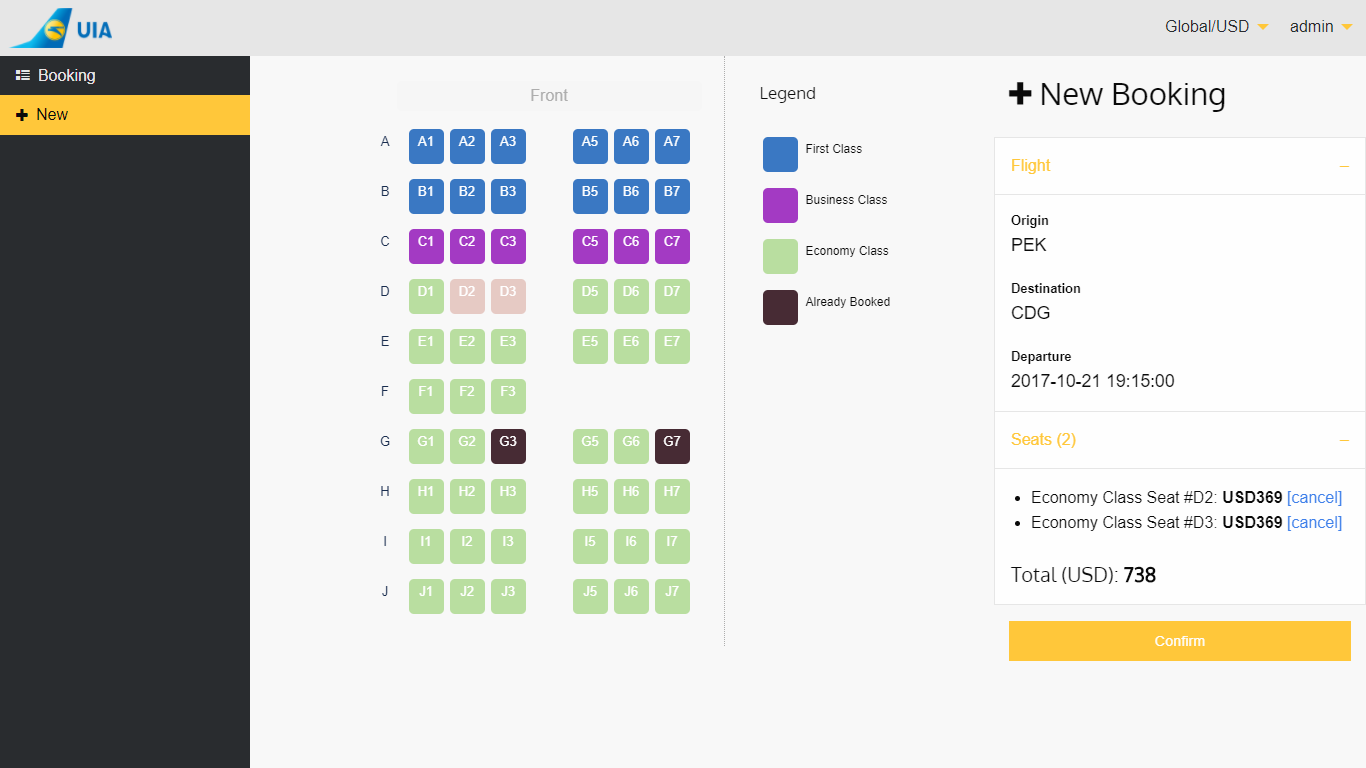


All of the project files were hosted and their changes were tracked using GitHub. 2 branches were created for the said project repository, where each branches are used for web applications of different region (e.g.: geo-us for US region web application). The link for the GitHub repository can be found in appendix section.

Several screenshots from the final system are shown as follow:

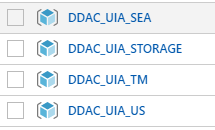






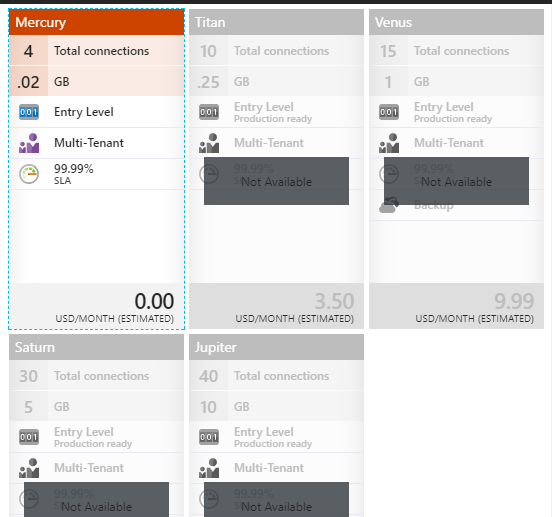
The demonstration of the finished system can be found in the video presentation provided in the appendix section.

## Azure Publishing

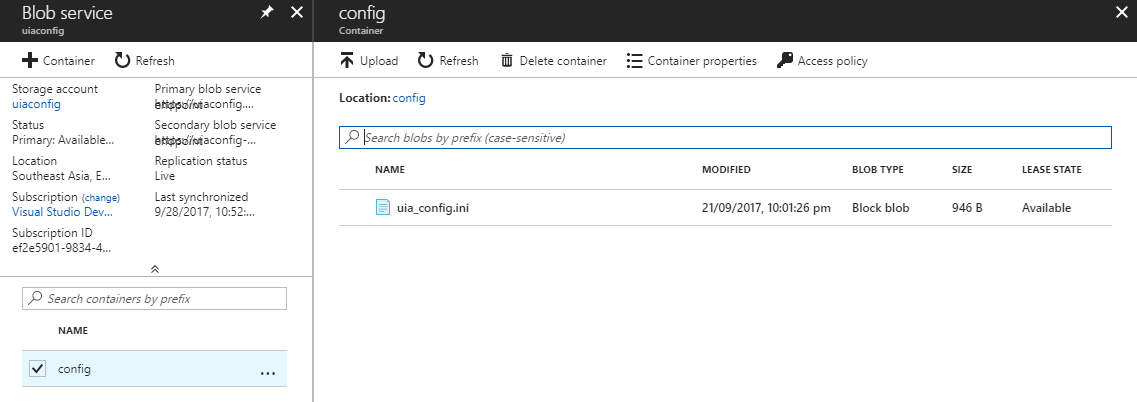
In order to get the web applications running on Azure, the app services, database, configuration file and traffic manager have to be created and published. Resource groups are first created to contain the required services.

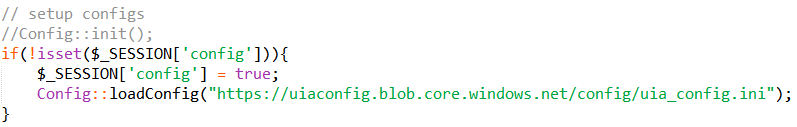
The resources which will be deployed in each resource groups are shown as follow:

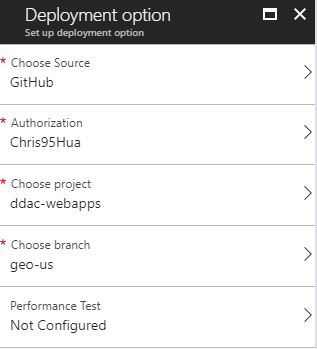
|  |  |
| --- | --- |
| **Resource group name** | **Resources** |
| DDAC\_UIA\_SEA | App service (SEA region) |
| DDAC\_UIA\_STORAGE | Storage Account and MySQL database (ClearDB) |
| DDAC\_UIA\_TM | Traffic Manager |
| DDAC\_UIA\_US | App service (South Central US region) |

The first resource that was created was the ClearDB’s MySQL database. Mercury pricing tier was chosen as it is the only pricing tier that was made available when creating the database. Also, since the application is not going to be used in product environment yet, thus sticking with Mercury pricing tier was a sensible choice as it is sufficient for a testing environment. It is worth noting that more pricing tiers were made available after creating the database service, therefore it is possible to scale the database up later on if there is a need to do so. The connection settings of the database were then acquired and used in uia\_config.ini, the application configuration file. The local database which was created during development phase was then migrated to the ClearDB database using MySQL Workbench.

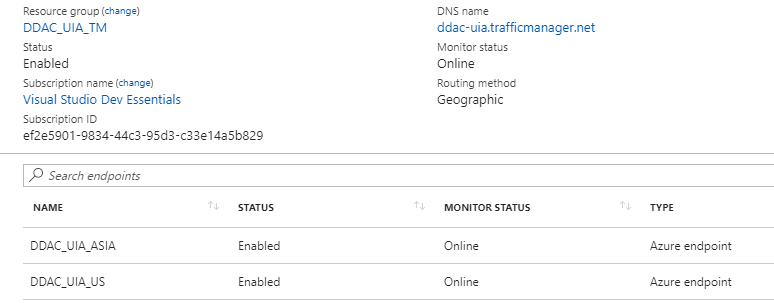
Apart from MySQL database, another resource that was created in the DDAC\_UIA\_STORAGE resource group was storage account. The storage account allows files and data to be hosted and retrieved whenever needed. In this case, it will be used to store uia\_config.ini, which is the web application configuration file. To do so, a container has to be created for the blob service before the web configuration file can be uploaded. Once the upload of the configuration file has been completed, the URL of the uploaded configuration file was then hardcoded in the setup.php of the web application so it can retrieve and read the file from the online storage.



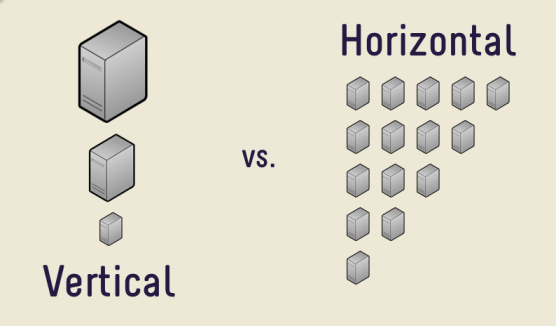


After getting the database and configuration file deployed, app services are created and deployed in SEA and South Central US regions. Both app services are using Standard S1 pricing, which is the minimum pricing tier that supports auto-scale and traffic manager. Since projects have been uploaded to GitHub, they can be easily deployed by setting up continuous deployment for the app services.

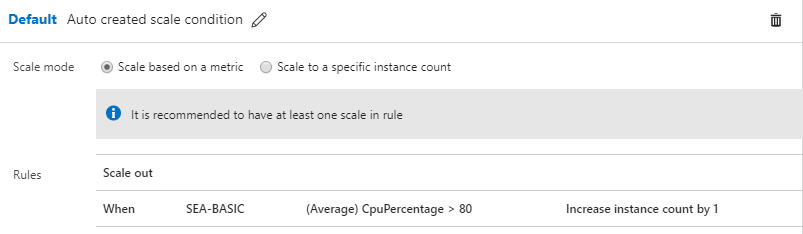
The last resource to be created is Traffic Manager. The URL of the app services were configured as the endpoints for the Traffic Manager.

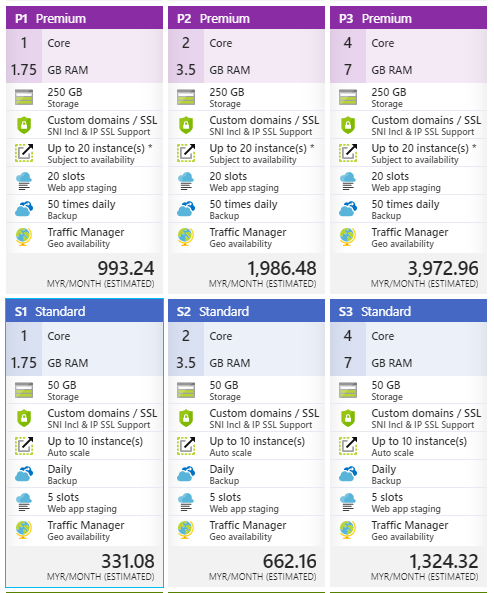


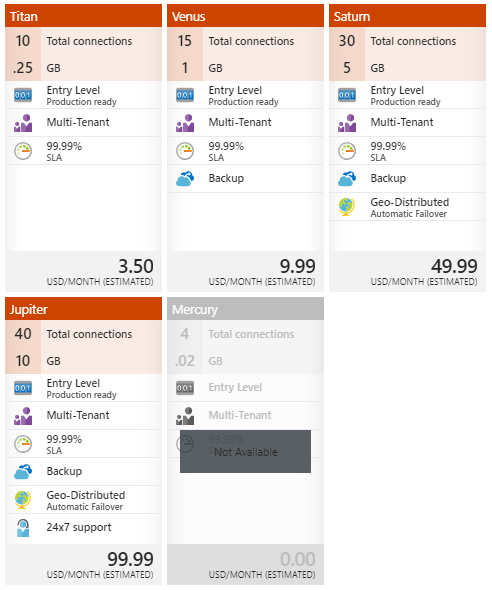
## Application Scaling

Azure’s app service can be scaled vertically (scale up) and horizontally (scale out). While both scaling methods help to improve the responsiveness of the application, the way that they achieve performance improvement are fundamentally different. Vertical scaling deals with the capacity of the resource, such as its storage size, CPU clock speed and RAM capacity (Microsoft, 2017). On the other hand, horizontal scaling deals with the instances of the resource by duplicating or removing the copies of the app services.

The app services created for this project were configured to scale out on metric. The instance of the web application will be increased by 1 automatically whenever the CPU load is above 80% on average for at least 15 minutes. Surely, more instances can be created as long as it does not exceed the limit set by the pricing tier. However, the instance count was kept low here as it is only used for testing. Also, since the CPU load actually never hits above 80% even with 1000 concurrent users, there is no point of requiring more instances, unless it is in production environment where the user load may spike up unexpectedly.



The app services can also be scaled up manually whenever necessary. Assume the performance offer by the current pricing tier, Standard S1 does not suffice, UIA can opt for other higher pricing tier such as Standard S2 or even Premium pricing tiers if they have the budget to do so. This scaling method is best used if UIA knew the web sites will have high traffic on certain period of time, such as during the holiday season or when they are doing promotions. Scaling up the app services will certainly improve the responsiveness of the app services, in addition to enabling more app instances to be used when scaling out.

Solely scaling up the application can only increase the performance so much if the database used by the app services does not get scaled up. Thus, in order to avoid bottleneck, it is recommended to scale up the MySQL database too if UIA wish to scale up the web services.

## Testing

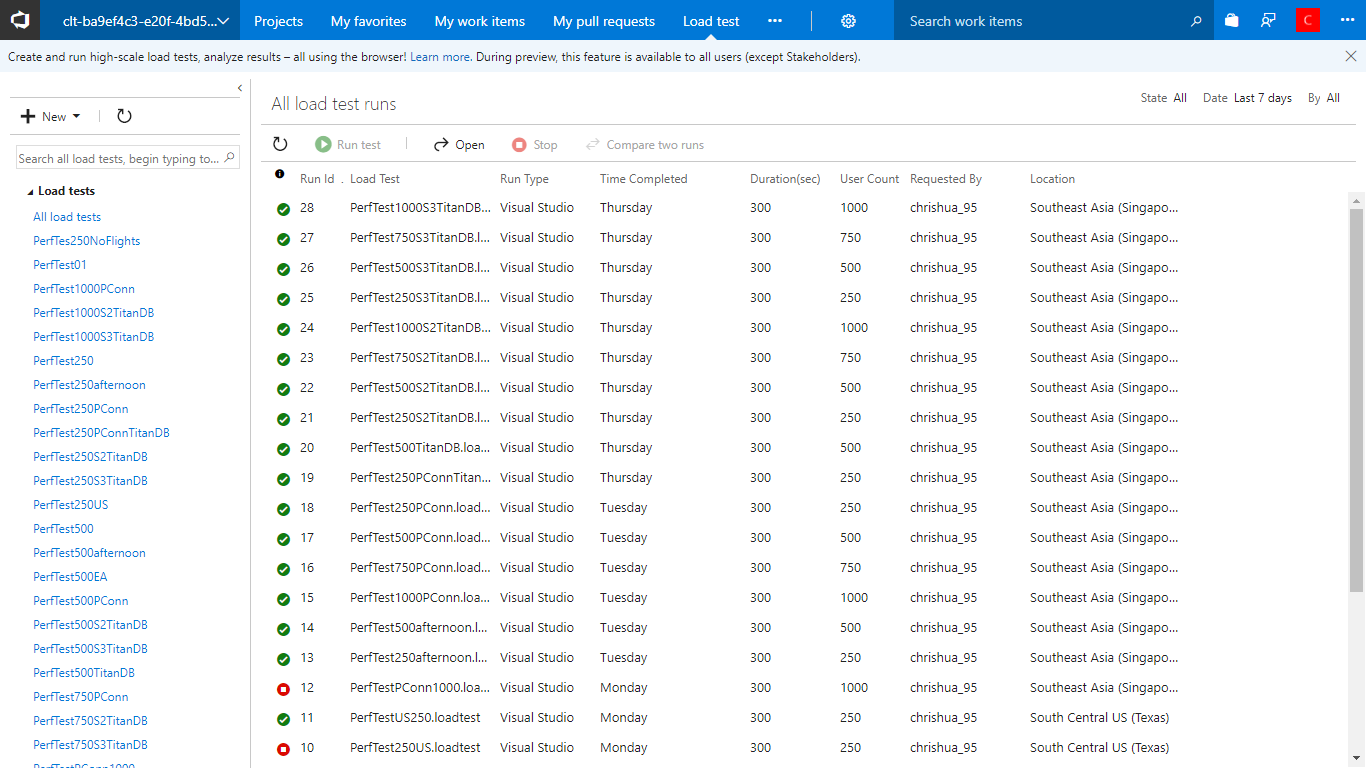
### Unit Testing

Unit testing has been conducted to verify the web applications are functioning as expecting. The results of the tests are shown as follow:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Title** | **Description** | **Expected result** | **Actual result** |
| **General site test suite** | | | | |
| GNL\_01 | 403 forbidden | Notify user when viewing page that is not permitted  **Procedure:**   1. While not logged in, enter “[base url]dashboard.php?page=main” in URL | Redirect to error 403 page | As expected |
| GNL\_02 | 404 not found | Notify user when invalid link is entered  **Procedure:**   1. Enter “[base url]/cookies.php” in URL | Redirect to error 404 page | As expected |
| GNL\_03 | Change currency | Switch the currency to the newly selected currency and update all pricings displayed  **Procedure:**   1. Hover over the “[region]/[currency]” menu and click on the desired currency | Switch the currency and update the displayed pricing | As expected |
| GNL\_04 | Navigate back to the main page of dashboard | Verify the main page of dashboard can be accessed from all valid pages once user is logged in  **Procedure:**   1. Sign in with a valid account 2. Click on “Booking” button from the side menu on the left | Redirect to the main page of dashboard | As expected |
| GNL\_05 | Navigate to search flight page | Verify search flight page can be accessed from all valid pages once user is logged in  **Procedure:**   1. Sign in with a valid account 2. Click on “New” button from the side menu on the left | Redirect to search flight page | As expected |
| GNL\_06 | Logout as signed in user | Clear user session and redirect user to index page  **Procedure:**   1. Sign in with a valid account 2. On the top menu bar, hover over name and click on “Logout” button | Redirect user to index page | As expected |
| **Index page test suite** | | | | |
| IND\_01 | Navigate to sign in page | Verify sign in page can be accessed from index page  **Procedure:**   1. Click on “Sign In” button | Redirect to sign in page | As expected |
| IND\_02 | Navigate to sign up page | Verify sign up page can be accessed from index page  **Procedure:**   1. Click on “Sign Up” button | Redirect to sign up page | As expected |
| **Sign in page test suite** | | | | |
| SIP\_01 | Navigate to sign up page | Verify sign up page can be accessed from sign in page  **Procedure:**   1. Click on “Sign up” hyperlink located below the sign in form | Redirect to sign up page | As expected |
| SIP\_02 | Sign in with invalid account | Nullify login attempt and display error message to user  **Procedure:**   1. Navigate to sign in page 2. Input invalid email address and password 3. Click on “Sign In” button | Unable to sign in and web page displays an error message indicating account is not created | As expected |
| SIP\_03 | Sign in with invalid password | Nullify login attempt and display error message to user  **Procedure:**   1. Navigate to sign in page 2. Input valid and registered email address but incorrect password 3. Click on “Sign In” button | Unable to sign in and web page displays an error message indicating password is incorrect | As expected |
| SIP\_04 | Sign in with valid account and password | Load user info and save into user session and redirect user to dashboard page  **Procedure:**   1. Navigate to sign in page 2. Input valid login credentials of a registered account 3. Click on “Sign In” button | Redirect user to dashboard page | As expected |
| SIP\_05 | Sign in with user session | Load user info using the saved session data and redirect user to dashboard page  **Procedure:**   1. Navigate to sign in page 2. Input valid login credentials of a registered account 3. Check “Remember me” checkbox 4. Click on “Sign In” button 5. Close the browser and navigate to the website again | Redirect user to dashboard page | As expected |
| **Sign up page test suite** | | | | |
| SUP \_01 | Navigate to sign in page | Verify sign in page can be accessed from sign up page  **Procedure:**   1. Click on “Sign in” hyperlink located below the sign up form | Redirect to sign in page | As expected |
| SUP\_02 | Sign up with invalid credentials | Request user to correct their credentials before submitting the credentials  **Procedure:**   1. Navigate to sign up page 2. Input invalid credentials 3. Click on “Sign Up” button | Display warning message and highlight the invalid input field(s) | As expected |
| SUP\_03 | Sign up with existing email | Request user to correct their credentials before submitting the credentials  **Procedure:**   1. Navigate to sign up page 2. Input an email address which has been registered on the website 3. Click on “Sign Up” button | Display error message indicating email address has already been used | As expected |
| SUP\_04 | Sign up with valid credentials | Save the user account into database and notify user  **Procedure:**   1. Navigate to sign up page 2. Input valid credentials 3. Click on “Sign Up” button | Display a success message indicating the account has been created successfully | As expected |
| **Dashboard page test suite** | | | | |
| DBD\_01 | Navigate to search flight page | Verify search flight page can be accessed from dashboard  **Procedure:**   1. Click on “New” button on top-right of the screen | Redirect to search flight page | As expected |
| DBD\_02 | View bookings summary | Load and display a list of flights booked by user  **Procedure:**   1. Navigate to dashboard page | Display a list of flights booked by user in table form | As expected |
| DBD\_03 | Booking pagination | Load another set of bookings summary with the given page offset  **Procedure:**   1. Navigate to dashboard page 2. Click on one of the pagination control buttons | Display a list of flights booked by user in table form with given page offset | As expected |
| **Booking detail page test suite** | | | | |
| DTL\_01 | View booking details | Load and display selected booking details  **Procedure:**   1. Navigate to dashboard page 2. Click on the Booking ID in the table | Display the booking and flight details of the given booking ID | As expected |
| **Search flight page test suite** | | | | |
| SCH\_01 | Select origin or destination | Remove the current selected value from another combo box  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin the combo box 3. Click on the destination combo box | The selected location from origin combo box is not present in destination combo box | As expected |
| SCH\_01 | Select flight routes | Load and display list of available departure dates for the given flight  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box | Display the list of available departure dates for the given flight | As expected |
| **Booking page test suite** | | | | |
| BKG\_01 | View flight | Load and display details of the selected flight for booking  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box 3. Select the desired departure time and click on “Search” button | Display selected flight details and list of seats available | As expected |
| BKG\_02 | Select seats | Display list of selected seats and update the total price  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box 3. Select the desired departure time and click on “Search” button 4. Select desired seats | Show list of selected seats along with their price as well as update the total price | As expected |
| BKG\_03 | Remove selected seats | Remove the selected seats and update the total price  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box 3. Select the desired departure time and click on “Search” button 4. With at least 1 seat selected, remove the selected seats by clicking on them or clicking on the “Remove” button | Remove the selected seats and update the total price | As expected |
| **Checkout page test suite** | | | | |
| CHK\_01 | View booking summary | Display a summary of the booking details which include selected seats, price and flight details  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box 3. Select the desired departure time and click on “Search” button 4. With at least 1 seat selected, click on “Confirm” button | Display the booking summary | As expected |
| CHK\_02 | Checkout and create booking | Save the booking and redirect user to the dashboard home page  **Procedure:**   1. Navigate to search flight page 2. Select the desired origin and destination locations from the combo box 3. Select the desired departure time and click on “Search” button 4. With at least 1 seat selected, click on “Confirm” button 5. Fill in the necessary credentials and click on “Checkout” button | Redirect user to dashboard home page | As expected |

### Performance Testing

A total of 26 load tests have been run to test the performance of the web applications on different configuration, pricing tier, user load and time of day to determine if the mentioned factors or aspects will affect the response time as well as requests per second of the web application. All of the tests will be run on the SEA app service with traffic generated from SEA region for 5 minutes.



The table below shows the summary of the load tests. Do note that some of the test name shown here had been modified to reflect the context and settings of the test more accurately. A detailed analysis will be presented in the following section.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Name** | **User Load** | **Avg Response Time (ms)** | **Req/sec** | **Successful** | **Failed** | **Date/Time** |
| PerfTest1000S3TitanDBPConn | 1000 | 522.1 | 1405.8 | 421739 | 0 | 9/29/2017 12:31:08 AM |
| PerfTest750S3TitanDBPConn | 750 | 211 | 1959.31 | 587793 | 0 | 9/29/2017 12:19:19 AM |
| PerfTest500S3TitanDBPConn | 500 | 246.9 | 1341.69 | 391600 | 10907 | 9/29/2017 12:07:29 AM |
| PerfTest250S3TitanDBPConn | 250 | 63 | 970.6 | 291118 | 63 | 9/28/2017 11:55:41 PM |
| PerfTest1000S2TitanDBPConn | 1000 | 718.2 | 1158.91 | 347672 | 0 | 9/28/2017 11:41:04 PM |
| PerfTest750S2TitanDBPConn | 750 | 542.6 | 1082.19 | 324658 | 0 | 9/28/2017 11:29:03 PM |
| PerfTest500S2TitanDBPConn | 500 | 321.1 | 1075.26 | 322578 | 0 | 9/28/2017 11:13:49 PM |
| PerfTest250S2TitanDBPConn | 250 | 57.6 | 1024.26 | 307263 | 15 | 9/28/2017 11:00:52 PM |
| PerfTest500S1TitanDBPConn | 500 | 1170 | 505.23 | 151568 | 0 | 9/28/2017 10:47:56 PM |
| PerfTest250S1TitanDBPConn | 250 | 451.1 | 527.45 | 158234 | 0 | 9/28/2017 10:35:37 PM |
| PerfTest250S1PConn | 250 | 484.4 | 382.90 | 114857 | 0 | 9/26/2017 11:31:04 PM |
| PerfTest500S1PConn | 500 | 1151.4 | 492.90 | 147875 | 0 | 9/26/2017 11:17:59 PM |
| PerfTest750S1PConn | 750 | 1834 | 450.20 | 135066 | 0 | 9/26/2017 11:04:43 PM |
| PerfTest1000S1PConn | 1000 | 2318 | 514.00 | 154215 | 0 | 9/26/2017 10:50:49 PM |
| PerfTest500S1afternoon | 500 | 1895 | 231.10 | 69340 | 0 | 9/26/2017 2:31:35 PM |
| PerfTest250S1afternoon | 250 | 508.8 | 341.10 | 102342 | 0 | 9/26/2017 2:18:45 PM |
| PerfTest750S1PConn | 750 | 3262 | 221.00 | 66312 | 0 | 9/25/2017 9:54:14 PM |
| PerfTest500S1PConn | 500 | 1878 | 249.60 | 74872 | 0 | 9/25/2017 9:37:03 PM |
| PerfTest250S1PConn | 250 | 492.1 | 336.20 | 100857 | 0 | 9/25/2017 9:19:56 PM |
| PerfTest500S1Cached | 500 | 2219 | 188.60 | 56585 | 0 | 9/25/2017 6:37:03 PM |
| PerfTest250S1Cached | 250 | 824.9 | 173.60 | 52084 | 0 | 9/25/2017 10:54:32 AM |
| PerfTest500S1 | 500 | 3605 | 129.80 | 38927 | 0 | 9/25/2017 10:14:47 AM |
| PerfTest250S1 | 250 | 1927 | 112.80 | 33851 | 0 | 9/25/2017 10:01:36 AM |

### Test Analysis

The results of unit tests are rather positive, as all of the test cases and functions can be executed and performed as expected. Thus, one can expect this web application to be functional and usable since it does not contains any severe bugs or issues. However, more tests such as security tests and user acceptance tests are needed to ensure the web application is truly reliable and capable of meeting users’ expectations.

As for the performance test, the results are fascinating to say the least. The following comparisons and analysis are segregated into 4 parts, where each part will discuss the findings found using different set of configurations or environment.

#### Non-optimized vs. optimized application

The very first version of the web applications were tested with a user load of 250 and 500. The results were rather unacceptable as the response time was almost 2 seconds on average when there are 250 concurrent users. Pushing the user load count to 500 results in 3.6s of response time on average, which is pretty bad.

However, it was speculated that the unoptimized external configuration store pattern is the reason behind the bad performance. As the pattern was applied in such a way that it will load the configuration store whenever the user is making request to the web application, thus it will reload and reread the configuration file from storage account. The codes were later reworked to cache the configuration file for each users, and the results after the optimized codes are shown as follow:

As seen, caching the configuration file reduced the average response time by at least 1 second, which is a significant improvement. The web application is further optimized by using persistent connection for the MySQL database connection. Persistent connection allows the database driver to cache the connection, which technically should be faster than re-establishing the connection when needed. After redoing the tests, the average request time was further decreased as a result of using persistent connection.

Caching the configuration file and database connection decreased the average response time by 2x when compared to the very first version of the application. Since this configuration provides a significant improvement for the web application, it was kept and used in later testing.

#### Time of day

It is speculated that the response time measured by Azure includes page load time, which measures the time requires for the user browser to fully receive and load the requested page. Thus, the response time can be affected by the network load or users’ internet connection. Therefore, another set of performance tests were ran on the following day in afternoon to verify this, as network load is generally higher during afternoon especially it is weekdays because it is office hours where more people are using the internet. And indeed, the average response time is slightly slower in afternoon compared to night as seen on the following comparison.

#### Network connection

As discussed previously, the response time can be affected by the network load or bandwidth. Since several undersea cables have been damaged in early September and are causing internet slowdown in SEA region, higher response time was expected from the SEA app service (M, 2017). However, a rather significant improvement in response time was observed in the night of 26th September 2017, which is a day after commencing the first performance tests. There might be a small possibility that the damaged cables were fixed on that day, although highly unlikely as it requires about 6 weeks of time to repair them as reported by the news. A more logical reason would be the traffic was redirected to nearby working cable which has lower load for SEA region on that night. The results of the tests can be seen in the chart, where there is a considerable improvement across the tests especially with a user load of 500 with same web application configurations.

#### Pricing tier

Various pricing tier of the app services and ClearDB MySQL database was chosen for the following tests to determine the improvement in performance as well as the performance to money ratio.

The first component of the web application that was scaled up is the MySQL database, from Mercury (free) tier to Titan tier. The pricing tier of the app service was maintained at S1 for the tests to verify if the web application is bottlenecked by database performance. The results proved that there is not much of difference in terms of response time.

Albeit the response time was largely the same, the number of request per second served was higher when there is 250 users. However, the gap closes when there is 500 users.

Thus, it is can be safely said that the database wasn’t really bottlenecking the performance of the web application in any way when there is more than 250 users. Therefore, the performance of the server used to host the web app should be looked at instead.

In the following tests, the app service was scaled to S2 and S3, and both are paired with Titan pricing tier of ClearDB’s MySQL database. The S1 + Mercury pricing tiers were used as the baseline for the following tests. As seen, there is a huge improvement when moving from S1 + Mercury to S2 + Titan, where the average response time is reduced by at least 3 times. This results in less than a second of response time on average when serving 1000 concurrent users, which is considered as ideal. However, the performance gained from S2 to S3 was negligible.

The scale of improvement holds true when looking at the average request per second, where there is a big jump when moving from S1 + Mercury to S2 + Titan, but the improvement from S2 to S3 was rather small in comparison.

An interesting thing to note is that when scaling up, the web application seems to be more prone to failure. The most failed requests ever gotten in a test are during the S3 test with 500 user load, where there are around 3% of failed requests. However, such failure rate is still considered acceptable since majority of the users specifically 97% of them are getting served with success.

While there is an improvement when scaling up the app service, it has diminishing returns in performance per price paid when a certain level of load is reached. As seen from the previous results, where the improvement from S2 to S3 is rather small and unnoticeable for most of the time when there is 1000 concurrent users or less, even though the S3 is twice as expensive as S2. Thus, in order to investigate which pricing tier offers the best value for UIA, their price-performance ratio will be identified. There are 2 metrics to look at when determining the price-performance ratio, namely the number of requests per second can be served for a Ringgit and the average response time to Ringgit ratio. The formula to determine the said results are shown as follow:

Basically, having more requests/sec served per Ringgit translates to better value for money. As for the average response times to Ringgit ratio, the closer the result is to 1.0, the more value for money it is (e.g.: similar to current ratio, where 1:1 (1.0) is ideal). If the result is higher than 1.0, it means slow response time for the price paid (response time outweighed the price paid). However, if the result is lower than 1.0, it means high price paid for the response time (price paid outweighed the response time).

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Response Time To Ringgit Ratio** | < 1.0 | 1.0 | > 1.0 |
| **Description** | High price paid for the average response time | Price/response time balanced | Slow average response time for the price paid |

The results acquired using the formulas are presented in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Pricing Tier** | **User load** | **Avg requests per sec / Ringgit** | **Avg response time to Ringgit ratio** |
| S1 + Mercury (RM 331.08) | 250 | 1.157 | 1.463 |
| 500 | 1.489 | 3.4777 |
| 750 | 1.36 | 5.5394 |
| 1000 | 1.552 | 7.0013 |
| S2 + Titan (RM 676.94) | 250 | 1.513 | 0.0851 |
| 500 | 1.588 | 0.4743 |
| 750 | 1.599 | 0.8015 |
| 1000 | 1.711 | 1.0609 |
| S3 + Titan (RM 1339.10) | 250 | 0.725 | 0.047 |
| 500 | 1.001 | 0.1844 |
| 750 | 1.463 | 0.1576 |
| 1000 | 1.049 | 0.3898 |

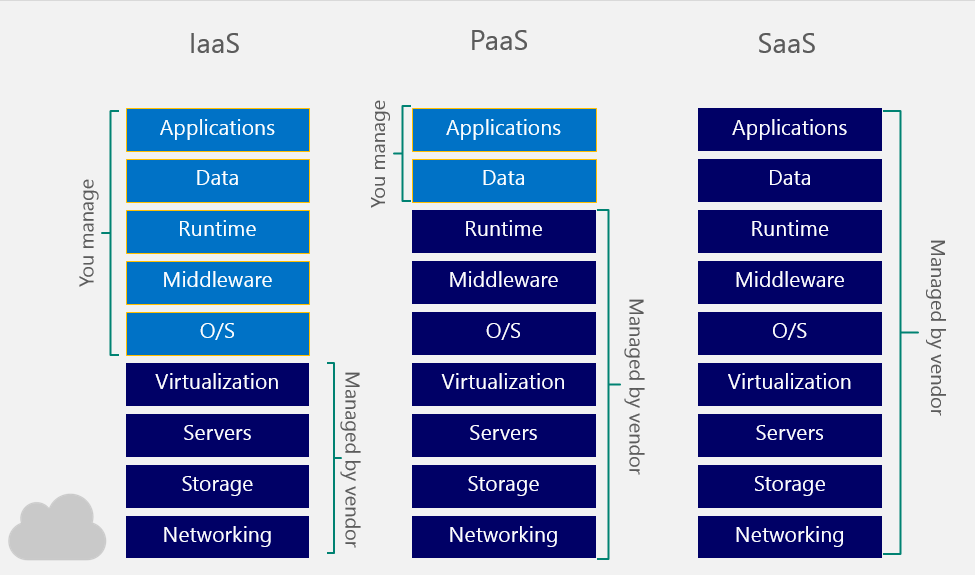
S2 + Titan pricing tiers are capable of serving more requests per second on average for a Ringgit. On the other hand, S3 + Titan have the worst average requests per second to price ratio.

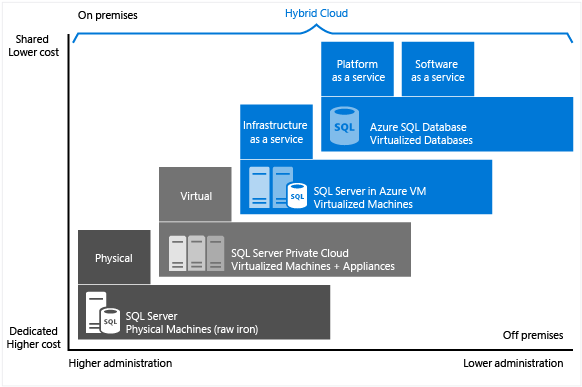
As seen from the chart below, the average response time to user load ratio of S2 + Titan pricing tiers are closer to 1.0 on average. S1 + Mercury often has a much higher value, making it the least ideal pricing tiers to be used as the average response time is way too slow for the paid price.

Therefore, based on the results shown, S2 + Titan pricing tiers offer the best price-performance ratio for UIA. However, the results may be different if the user load is more than 1000 since S3 + Titan pricing tier is technically more powerful and thus, it is speculated to offer better price-performance ratio when there is more concurrent users.

Conclusively, application optimization, time of day, network bandwidth, network status and the pricing tiers of app services and database are some of the factors that can affect the response time as well as request/sec of web application. Additionally, it has been found out that S2 + Titan pricing tiers have the best price-performance ratio for UIA, assume that they will only get a maximum of 1000 concurrent users during peak hour.

## Managed Database (PaaS)

 Cloud database is perhaps one of the more widely-used services on cloud platform such as Microsoft Azure and Google Cloud Platform because it offers plenty of advantages over traditional database while being affordable. The most popular type of cloud database is relational database as a service (DBaaS) service, essentially a managed database hosted on the cloud. Such database falls under the category of platform as a service (PaaS) as users are only required to manage the data themselves but not the platform that host the database because the latter was managed by the service provider. Most cloud service providers will also offer the choice to host database via virtual machines, basically providing infrastructure as a service (IaaS). This differs from PaaS database because the OS, DBMS and other software have to be installed or maintained by the users (Microsoft, 2017). Therefore, one can consider PaaS to be more user-friendly and hassle-free compared to IaaS as most of its aspects or components are managed automatically.



The diagram from Microsoft summarizes the cost, on/off premise, level of administration needed when hosting the database using different kind of methods. Unsurprisingly, PaaS database is the cheapest and most hassle-free option because of the aforementioned reasons. Taking this online flight booking system for UIA as example, the ClearDB MySQL database service was created with just a few clicks. Then, the database schema and data were migrated easily via MySQL workbench using the connection string provided. Since the platform was provided as service, a lot of processes were simplified and thus, setting up and running the database is relatively easy.

Indeed, one can argue that hosting database from dedicated servers offer more control and can be tailored to fit very specific needs. However, several benefits offered by PaaS database are unparalleled compared to other database hosting method. As mentioned, PaaS database is actually cheaper to setup and run due to the users do not have to acquire the necessary dedicated hardware themselves besides the resources needed to maintain the hardware and software were virtually eliminated. Additionally, the pay-as-you-go model is relatively more affordable as the users only have to pay for what they use. Not to mention that setting up and scaling the hardware can be done on-demand and quickly (Rose Business Technologies, 2011). PaaS database often also offer features like disaster recovery, firewall security, data encryption, database replication and geo-distribution, which are rather useful especially for those who are seeking for peace of mind when hosting their data but unwilling to go through all of the works to get those features running.

The nature of PaaS database made it suitable for teams which have a limited budget and have time constraints in development, especially smaller teams which cannot afford to hire more people to setup and manage the database for them. However, PaaS database lacks the amount of control found in database hosted on dedicated servers. Assume one really wants have more control over their database while still looking for a cost-effective solution, they can opt for IaaS database as it offers exactly that.

Some examples of fully-managed cloud database would be Microsoft Azure SQL Database and Google Cloud Platform’s Cloud SQL. These databases are capable of self-managing, such as automating the backups and replications, thus require near-zero maintenance. Albeit both PaaS databases offered by Google Cloud Platform and Microsoft Azure are powerful, highly accessible, scalable and available, they are quite limited in terms of database engine supports. As of current, Azure SQL Database only supports SQL Server Engine, with MySQL support coming soon (under Azure Database for MySQL, which is currently in Preview stage at the time of writing). On the other hand, Google Cloud Platform’s Cloud SQL supports MySQL 5.6 and 5.7, with support for PostgreSQL coming soon.

As for Amazon AWS Relational Database Service (RDS), it is leaning towards container service side due to the lack of self-managing capabilities found in the database solutions mentioned previously. Nevertheless, AWS RDS still offer plenty of features and it shares a lot of similarities with its competitors. AWS RDS has the upper hand of supporting more database engines, such as SQL Server, MySQL, ProgreSQL, Oracle and MariaDB. All of the said database engines are hosted by AWS on Elastic Block Store (EBS) volumes. They also have a 5th database engine called Aurora which uses different infrastructure and claims to be able to delivers up to 5 times the throughput of standard MySQL running on the same hardware. This is achieved by tightly integrating the database engine with an SSD-backed virtualized storage layer purpose-built for database workloads (Amazon, 2017).

The above comparisons and discussions only highlight some SQL database offered by the cloud platform providers and the difference between them. The presented points were obviously limited as it does not touch on other aspects such as their pricing, availability and locations. One should also consider about other products and services offered by them, such as NoSQL database services, storage and networking products when choosing which cloud platform providers to use for the projects, as sticking to one cloud platform provider is more convenient and easier to manage.

To summarize, PaaS database offers a lot of advantages over the traditional database, which can improve the application development, publishing and maintenance experience.

# Conclusion

Cloud computing platform has made a lot services more accessible besides making tasks simpler and cheaper to accomplish. Going through this project, it definitely shows how a cloud platform like Microsoft Azure can benefits a company and its development team. The tools and benefits offered can facilitate high quality products and experience to be delivered easily and at an affordable cost. The UIA online flight booking system is an example of such case, where it was developed, deployed and tested within a week of timeframe. All in all, cloud computing will continue and further change the landscape of software development in the future as the architecture and services matured as well as the cost gets more affordable.

# References

Amazon, 2017. *Amazon Aurora *. [Online] Available at: <https://aws.amazon.com/rds/aurora/details/> [Accessed 28 September 2017].

M, M., 2017. *Damaged undersea cables causing Internet slowdown*. [Online] Available at: <http://www.thestar.com.my/tech/tech-news/2017/09/04/damaged-undersea-cables-causing-internet-slowdown/> [Accessed 28 September 2017].

Microsoft, 2017. *Autoscaling*. [Online] Available at: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling> [Accessed 24 September 2017].

Microsoft, 2017. *Choose a cloud SQL Server option: Azure SQL (PaaS) Database or SQL Server on Azure VMs (IaaS)*. [Online] Available at: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-paas-vs-sql-server-iaas> [Accessed 29 September 2017].

Microsoft, 2017. *External Configuration Store pattern*. [Online] Available at: <https://docs.microsoft.com/en-us/azure/architecture/patterns/external-configuration-store> [Accessed 23 September 2017].

Rose Business Technologies, 2011. *Benefits of Platform as a Service (PaaS)*. [Online] Available at: <http://www.rosebt.com/blog/benefits-of-platform-as-a-service-paas> [Accessed 28 September 2017].

# Appendix

GitHub repository link: <https://github.com/Chris95Hua/ddac-webapps/>

Video presentation link: <https://web.microsoftstream.com/video/3f989db5-47c4-4842-a941-5d35d6f20b88>

Available account:

Email: admin@uia.com

Password: 123456