Final Report

14 October 2018

Version 1.0

CMSC 495

Jonathan Barnes, Collin Hicks, Matthew Orahood

University of Maryland University College

Table of Contents

[1. Overview 6](#_Toc527289279)

[**1.1.** **Roles and Responsibilities.** 6](#_Toc527289280)

[**1.2.** **Resource Requirements** 7](#_Toc527289281)

[**1.3.** **Application Function** 8](#_Toc527289282)

[**1.4.** **Development Requirements** 8](#_Toc527289283)

[2. Use Cases. 11](#_Toc527289284)

[3. Security Testing 12](#_Toc527289285)

[4. Key Performance Indicators Timeline. 12](#_Toc527289286)

[5. Risk Mitigation and Security Controls. 14](#_Toc527289287)

[6. Project Costs. 15](#_Toc527289288)

[7. Project Successes and Production Observations 16](#_Toc527289289)

[8. About this guide 16](#_Toc527289290)

[9. Use of Guide 16](#_Toc527289291)

[10. Definitions 17](#_Toc527289292)

[11. Carvue Key Components 18](#_Toc527289293)

[**11.1.** **Application Overview** 18](#_Toc527289294)

[**11.2.** **Development History** 18](#_Toc527289295)

[**11.3.** **Key Personnel** 18](#_Toc527289296)

[**11.4.** **Deployment** 19](#_Toc527289297)

[12. Security and Privacy Considerations 19](#_Toc527289298)

[13. Privacy Concerns 20](#_Toc527289299)

[14. Application Components 20](#_Toc527289300)

[15. What is Carvue? 21](#_Toc527289301)

[**15.1.** **Application Features** 21](#_Toc527289302)

[**15.2.** **Directory** 21](#_Toc527289303)

[**15.3.** **Environment Layout** 22](#_Toc527289304)

[16. Composition of Application 24](#_Toc527289305)

[16.1. Application Performance 24](#_Toc527289306)

[16.2 Application Health 24](#_Toc527289307)

[**16.3.** **Version control** 25](#_Toc527289308)

[**16.4.** **Cloud infrastructure** 25](#_Toc527289309)

[**16.5.** **Web Browser** 25](#_Toc527289310)

[**16.6.** **Escalation for Troubleshooting** 25](#_Toc527289311)

[17. System Use and Application Navigation 25](#_Toc527289312)

[**17.1.** **Application Operation** 25](#_Toc527289313)

[**17.2.** **User Interface** 26](#_Toc527289314)

[**17.3.** **Authorization and Authentication** 26](#_Toc527289315)

[18. Installing the Application 27](#_Toc527289316)

[19. Provisioning the Application 27](#_Toc527289317)

[20. Terminating Application Instance 28](#_Toc527289318)

[21. Application Use and Overview 28](#_Toc527289319)

[**21.1.** **System layout** 28](#_Toc527289320)

[**21.2.** **Error Handling Instructions** 29](#_Toc527289321)

[**21.3.** **Messages** 29](#_Toc527289322)

[**21.4.** **Reference Guide** 29](#_Toc527289323)

[22. Test Plan 30](#_Toc527289324)

[**22.1.** **Purpose** 30](#_Toc527289325)

[**22.2.** **Test Strategy** 30](#_Toc527289326)

[23. Project Overview 30](#_Toc527289327)

[**23.1.** **Audience** 31](#_Toc527289328)

[24. Test Strategy 31](#_Toc527289329)

[**24.1.** **Test Objectives** 31](#_Toc527289330)

[**24.2.** **Test Assumptions** 31](#_Toc527289331)

[**24.3.** **Test Principles** 32](#_Toc527289332)

[**24.4.** **Data Approach** 32](#_Toc527289333)

[**24.5.** **Scope and Levels of Testing** 32](#_Toc527289334)

[**24.5.1.** **Exploratory Testing** 33](#_Toc527289335)

[**24.5.2.** **Functional Testing** 33](#_Toc527289336)

[**24.5.3.** **User Acceptance Testing (UAT)** 33](#_Toc527289337)

[**24.5.4.** **Documents** 34](#_Toc527289338)

[**24.5.5.** **Execution Strategy** 34](#_Toc527289339)

[**24.6.** **Test Cycles** 35](#_Toc527289340)

[**24.7.** **Validation and Defect Management** 36](#_Toc527289341)

[**24.8.** **Test Metrics** 37](#_Toc527289342)

[**24.9.** **Defect tracking & Reporting** 37](#_Toc527289343)

[**24.10.** **TEST MANAGEMENT PROCESS** 38](#_Toc527289344)

[**24.10.1.** **Test Management Tool** 38](#_Toc527289345)

[**24.10.2.** **Test Design Process** 38](#_Toc527289346)

[**24.10.3.** **Test Execution Process** 39](#_Toc527289347)

[**24.11.** **Test Environments** 39](#_Toc527289348)

[**24.12.** **Approvals** 39](#_Toc527289349)

[25. Design Overview 40](#_Toc527289350)

[**25.1.** **Background Information** 40](#_Toc527289351)

[**25.2.** **Application Migration** 41](#_Toc527289352)

[**25.3.** **Current Application Hosting** 41](#_Toc527289353)

[**25.4.** **Technology Forecast Integrations** 41](#_Toc527289354)

[**25.5.** **Constraints** 41](#_Toc527289355)

[26. Flask 42](#_Toc527289356)

[27. Stripe Integration for Payments 43](#_Toc527289357)

[28. AWS for Reliability 43](#_Toc527289358)

[29. User Characteristics 43](#_Toc527289359)

[30. User Challenge 44](#_Toc527289360)

[31. User Requirements and Needs 44](#_Toc527289361)

[32. System Architecture 44](#_Toc527289362)

[**32.1.** **Hardware Architecture** 44](#_Toc527289363)

[**32.2.** **Software Architecture** 45](#_Toc527289364)

[33. Network Architecture 50](#_Toc527289365)

[34. AWS Architecture 51](#_Toc527289366)

[35. Data Design 52](#_Toc527289367)

[**35.1.** **PostgreSQL Schema** 52](#_Toc527289368)

[**35.2.** **ERM Logic Model** 53](#_Toc527289369)

[**35.3.** **Database Schema** 53](#_Toc527289370)

[**35.4.** **Non-Database Management System Files** 55](#_Toc527289371)

[36. Detailed Design 57](#_Toc527289372)

[37. Hardware Detailed Design 57](#_Toc527289373)

[38. Software Detailed Design 58](#_Toc527289374)

[**38.1.** **Registrations Module (mod\_registrations)** 58](#_Toc527289375)

[**38.1.1.** **Processing** 58](#_Toc527289376)

[**38.1.2.** **Local data structures** 59](#_Toc527289377)

[**38.2.** **Inventory Module (mod\_inventory)** 60](#_Toc527289378)

[**38.2.1.** **Processing** 60](#_Toc527289379)

[**38.2.2.** **Local data structures** 61](#_Toc527289380)

[**38.3.** **Sessions Module (mod\_sessions)** 61](#_Toc527289381)

[**38.3.1.** **Processing** 62](#_Toc527289382)

[**38.3.2.** **Local data structures** 62](#_Toc527289383)

[**38.4.** **Purchases Module (mod\_purchases)** 63](#_Toc527289384)

[**38.4.1.** **Processing** 63](#_Toc527289385)

[**38.4.2.** **Local data structures** 64](#_Toc527289386)

[39. Communications Detailed Design 65](#_Toc527289387)

[40. Communications Architecture Overview 66](#_Toc527289388)

[41. External Interface Design 66](#_Toc527289389)

[**41.1.** **Stripe** 66](#_Toc527289390)

[**41.2.** **Stripe Detailed Design** 68](#_Toc527289391)

[**41.3.** **Human-Machine Interface** 68](#_Toc527289392)

[**41.4.** **Interface Design Rules** 68](#_Toc527289393)

[**41.5.** **Inputs** 68](#_Toc527289394)

[**41.6.** **Outputs** 69](#_Toc527289395)

[**41.7.** **Navigation Hierarchy** 69](#_Toc527289396)

[**41.8.** **User Registration** 69](#_Toc527289397)

[**41.9.** **Session Creation** 69](#_Toc527289398)

[**41.10.** **Inventory Listing** 70](#_Toc527289399)

[**41.11.** **Vehicle Details** 70](#_Toc527289400)

[**41.12.** **Vehicle Purchase** 71](#_Toc527289401)

[42. System Integrity Controls 71](#_Toc527289402)

[43. Verification for CRUD operations 72](#_Toc527289403)

[44. Security Control Tests 73](#_Toc527289404)

[45. Conclusion 74](#_Toc527289405)



The following usernames and passwords can be used to test the application:

* Non-admin: test\_user@example.com / password
* Admin: test\_admin@example.com / password

# 1. Overview

This project is designed to be a standalone production web application for the CMSC 495 capstone course called “Carvue”. The team will design, develop, and run a simple web application that is built upon the Flask framework using Python scripts to execute routes and codes. The application will utilize a MySQL database and schema to support the application. It will have simple CRUD (create, read, update, destroy) functionality and the theme will be that of a car buying website. The project will require the role of three contributors. A team lead with full stack development responsibilities, infrastructure and cloud architect, and a database developer and administrator will each contribute to the project.

## **1.1. Roles and Responsibilities.**

The below tables will detail the roles and responsibilities of each project contributor. This project is divided into essentially three roles. One, a full stack developer who also conducts security testing, an infrastructure and cloud architect, and a database administrator. The detailed overview of each is provided below.

a. **Collin Hicks** - Full Stack Development and Team Leader. Responsible for developing within Flask Framework and Python Script. Will also run Security Tests to ensure application security.

b. **Matthew Orahood** - Infrastructure and Cloud Architect. Responsible for maintaining virtual environments for application hosting and version control. Provides quality assurance and assist in any full stack development roadblocks. Ensures AWS Elastic Bean Stalk and AWS resources are provisioned and running. Also responsible for version control.

c. **Jonathan Barnes** - Database Administrator. Responsible for database schema, database migrations, and defining fields for users and roles for the web application.

## **1.2. Resource Requirements**

This project will require several key resources for success. The framework that the application is built on is Flask, which incorporates important routing and template functions. The Python code will be written in several IDEs, including Atom and PyCharm. In terms of hosting the application that will be done through AWS Elastic Beanstalk. This is an efficient means of deploying a fully scaled web application. AWS Elastic Beanstalk allows us to upload our code from the Git repo and AWS automatically provisions the necessary hardware and supporting infrastructure. The below table details the project resources and the other components required for project completion.

|  |  |
| --- | --- |
| Project Resources | |
| Type | Name |
| Hardware | iMac 2015, Comodity Servers via AWS |
| Software | Atome IDE, PyCharm |
| Operating System | Linux, Microsoft Windows 10, Apple macOS High Sierra |
| Languages | Python, MySQL |
| Frameworks | Flask, Jinja2, SQL Alchemy |
| Version Control | Git |
| Production Resources | AWS Elastic Beanstalk stack. |

## **1.3. Application Function**

This section details the functional requirements for the application. This includes: register, login, create, update, and delete functions. The application is designed to function as a clone of a popular car research and purchasing web application. The system is designed to allow a user to register online. Authentication is then enforced on a separate login page that takes a username and password. Upon successful login, the user is presented a homepage with a dashboard of available cars. The user can then select a car and is routed to a car specific page that lists details such as make, model, and year. The user will then be able to select the item for purchase. The purchase will be a simple update to the database that lists the car as now purchased. After purchase, the user can then logout of the application or continue shopping for vehicles.

## **1.4. Development Requirements**

The below section will break down the main components to be addressed within development. The development process will involve several major steps to ensure the project is successfully implemented. These processes include defining the users and roles, defining the outcomes of CRUD (create, read, update, delete) functions, and the outcome of template pages. The below subsections will provide further detail.

1. **Users**

There are three user types:

1. Admin. User will have root credentials to the application to add, delete, and update listed cars. Will also be able to delete users.
2. Customer1. User will be able to login, see car selection dashboard, and purchase a vehicle.
3. Customer2. User will be able to login, see car selection dashboard, and purchase a vehicle.

**B. Authentication**

Authentication will be handled via username and password credentials. Passwords will be hashed and no attributable.

**C. Client**

The application will be accessible on most major web browser clients (i.e. Chrome, Safari, Firefox).

**D. Production Server**

The production server is generated and maintained via AWS Elastic Bean Stalk. The

**E. Version Control**

Git will be used for version control with two branches.

i. Master Branch.

ii. Development Branch.

**F. CRUD Operations**

a. Create Vehicle

i. Admin is able to login, create a post that will be displayed to dashboard, that shows vehicle for sale.

ii. Will include vehicle picture, price, and make/model upon display.

iii. Will include button to select which redirects to vehicle specific page.

b. Read Vehicle

i. Users will be able to see vehicles displayed on dashboard.

ii. Vehicle will have its own page that displays:

1. Vehicle Name

2. Vehicle Year

3. Vehicle Make

4. Vehicle Model

5. Vehicle Mileage

6. Vehicle Price

7. Update Vehicle

iii. The admin user can update vehicle information.

iv. The customer user can select vehicle for purchase which is then reflected in SQL database.

c. Delete Vehicle

i. The admin user can delete the vehicle from the application.

ii. Must be masked from customer user types.

**G. Purchase**

The customer user will be able to select a vehicle for purchase. This will then place a vehicle in a cart to fill out data for purchase.

a. Name

b. Address

b. Credit Card

**H. Login**

* 1. A login tab will be present on the homepage.

**I. Register**

1. A register tab will be present on the homepage.

b. It will redirect the user to a register page.

c. Once registered the user is directed to login page.

**J. Dashboard**

a. The dashboard page will display all vehicles with access to delete functions based on root or non-root access. Each vehicle will contain a button to redirect to vehicle’s own detailed page.

**K. Vehicle Page**

a. The vehicle page will display pertinent vehicle data. Customer user will interact with this page to purchase.

# 2. Use Cases.

In the below use cases we will detail how the two user types (the administrator and customer) interact with the application. Furthermore, we will detail the security use case and what requirements are involved to properly test the security controls. For the purposes of this project there will be two users and three specific use cases. The user types and their respective use cases are detailed below.

1. Admin Login.

This use case will have an admin user type login to the application. The user will then upload several vehicles of different makes and models. The user will view the vehicles from the dashboard page. The admin user will select a vehicle and delete it from the application. Finally, the admin will logout.

II. Customer 1 Login

In this use case the customer will register for the application. After registering, they will login. At the dashboard page they will select a vehicle and navigate to its page. They will purchase the vehicle and logout.

III. Customer II Login

In this use case we will test the authentication of the application. The user will attempt to login without registering and be rejected. The user will then register, but will attempt to login with invalid credentials and be rejected. User will attempt to navigate to DELETE route but will not be able to access this route and its associated views.

# 3. Security Testing

Run manual tests for the following:

1. Cross Site Scripting Attack.

2. SQL Injection Attack.

3. Leakage Information.

4. Fingerprint Web Server.

Run automated tests for the following:

1. AWS Inspector.

# 4. Key Performance Indicators Timeline.

The below timeline is divided into week increments. Within each week there are several key indicators of completion. In terms of project tracking, our team utilized a Gantt Chart for project visualization as well as a Trello board from scrum sprints.

Week 1 (September 2 – September 9) Development Creation & Initial Routes

* Define Project Routes-
* Get, Post, Delete, etc.
* Create Python virtual environment
* Install Flask
* Install SQL Alchemy
* Jijna2 extension
* Create AWS Elastic Bean Stalk Instance
* Create Git Repo
* Push test to Git.
* Create Dev and Master Branches.

Week 2 (September 10- September 17)

* Create Templates
* Create config files
* Create SQL Schema
* Create SQL Tables
* Configure Python dev environment to SQL Schema

Week 3 (September 17- September 24)

* + - Test Templates
    - Seed DB
    - Create Vehicle Landing Pages
    - Test Vehicle Dashboard

Week 4 (September 24 - October 1)

* Finish full stack development.
* Push application to development branch on Git.
* Test application push to AWS Elastic Beanstalk.

Week 5 (October 1 - October 7)

* Begin security testing of application.
* Begin quality assurance for usability.
* Test database migrations.

Week 6 (October 7 - October 14)

* Validate security controls.
* Push to master branch of git.
* Deploy application and monitor application health.
* Project completion.

# 5. Risk Mitigation and Security Controls.

The below section will detail security controls that will be addressed within the flask framework. The application has security controls in regards to session management, role-based access, and authentication. As this is a web application the main security concern revolves around web-based vulnerabilities, such as not stripping special characters in login fields, and the main security controls address these issues (Flask, 2018).

|  |  |  |  |
| --- | --- | --- | --- |
| Security Controls | | | |
| Security Control | Framework | Feature | Vulnerability & Threat Mitigation |
| Session Authentication | Flask | Session authentication via Flask-Login. | Session Hijacking Attempts |
| Role Based Access | Flask | Assign admin, editor, SuperUser roles to user. | Privilege escalation. |
| Password Hashing | Flask |  | Brute Force Attacks, Birthday Attacks, Credential theft. |

# 6. Project Costs.

The costs for this project will be minimal due to leveraging open source frameworks such as Flask alongside Amazon Web services to host the production application. The estimated cost for hosting this application will be zero dollars as we can apply our AWS education credit to the production server (AWS, 2018). Also, PyCharm and Atom IDE have free options that allow us to complete full development in a virtual python environment. The other resources, such as Git and OWASP are also free and require no cost. The below sections will detail associated costs with each component of the application.

|  |  |  |
| --- | --- | --- |
| Resource Costs | | |
| Resource | Projected Costs | Additional Expected Costs |
| AWS Elastic Beanstalk | $0.00 | The education credit is valued at $200.00. If the production application resources go over that amount we will utilize the free tier of Elastic Beanstalk. |
| Atom IDE, PyCharm | $0.00 | None |
| Operating Systems | $0.00 | None |
| Version Control | $0.00 | None |
| Frameworks | $0.00 | None |

# 7. Project Successes and Production Observations

This section describes the application behavior and success criteria after it has moved from development to production. The criteria will be defined before development and will center upon several key factors including, application performance, user experience, and security control testing. The below table will define the success criteria and expected outcomes.

# 8. About this guide

This document is the final draft version of the Carvue user guide application. The final draft will include updated screenshots of the fully built website. This stage of the draft is designed to show initial definitions, operating capabilities, and intended use of this web application. The guide is meant to be used by an organization that wishes to post vehicles to the application and use the service to sell their vehicles.

# 9. Use of Guide

Use of this guide is limited to two types of users:

1. Customer - This is defined as a user who logins into the application to purchase and or view vehicles. They have read and list access.

2. Administrator - This is defined as the user who maintains the Carvue web application. The administrator can add, delete, and edit vehicles. This user has root privileges of the application to include read, list, delete.

# 10. Definitions

* Definition of Project Syntax
  + All code information will be written in plain text.
  + All syntax will be language dependent. This will be recognized by looking at the file extension if any code id displayed.
* Screenshots will be in one standard form.
* Definitions of Project Components
  + Flask - A small but extensible micro-framework, with several extensions, that is written in python.
  + Python - A general purpose programming language that relies upon simplicity, readability, and a standard library that helps create a strong and flexible means of programming.
  + Amazon Web Services- An on-demand cloud computing platform.
  + AWS Elastic Beanstalk – An AWS service that deploys web applications by uploading code to the service. Elastic Beanstalk ensures capacity, provisions back end resources, and is pay as you go like other AWS services.
  + PostrgreSQL – An object-relational database system utilized to store data.

# 11. Carvue Key Components

## **11.1. Application Overview**

The purpose of this section is to detail and describe the functionality and use of the Carvue web application. This document is the standardized reference guide for use of the system. The document link will be attached to the main web page. The below sections highlight the history of its development, the project owners, planned deployment, and other relevant documents.

## **11.2. Development History**

This project was conceived by Jonathan Barnes, Collin Hicks, and Matthew Orahood to act as a clone of the popular Caravana web application. This project is designed to be a capstone project for CMSC 495 at University of Maryland, University College. The design was planned to highlight the flexibility of newer frameworks within web development while also providing a way to test security functions and application health. The development of this application took place within the Flask framework utilizing Python,PostgreSQL, Jinja2, and other components to deliver a fully functional web application. It is hosted on AWS Elastic Beanstalk and maintained by Matthew Orahood as the infrastructure architect.

## **11.3. Key Personnel**

* Collin Hicks- Full Stack Developer and Team Leader.
* Jonathan Barnes- DB Administrator and Architect.
* Matthew Orahood - Cloud Architect and Infrastructure Architect, Flask development.
* Clients
* General Public. The site will be available through a dedicated EBS domain URL. The site will also be posted to a public Git Hub repository for open source feedback.

## **11.4. Deployment**

Cloud Deployment. The web application will be hosted via AWS Elastic Bean Stalk. The URL is:<http://team-flask-app-dev.us-east-2.elasticbeanstalk.com/inventory/>

Client Implementations. Currently there are no plans for a full client implementation. This is being offered as a free application for public consumption and academic research.

# 12. Security and Privacy Considerations

The development team took security into key consideration during development. The Carvue team viewed this as a development project with security as an integral component of the product and not as a simple check the box exercise. Care was taken to ensure that the application meets major security standards as described by industry and governmental standards. The below section details key security features within the application and the hosting environment.

* Web Application Security
  + Framework. The main framework utilized for this project is Flask. The Flask framework utilizes the following security features to secure the Carvue application.

a. Session Based Authentication to prevent session hijacking.

b. Role Based Access to ensure privileges are not escalated.

c. Password hashing to prevent brute force attacks.

d. User Registration to provide appropriate authentication and authorization.

e. Login tracking to provide log analysis if breaches occur.

d. CSFR keys are used to prevent cross site forgery request attacks.

* Cloud Security.

a. AWS. The application will be hosted on AWS Elastic Beanstalk. This AWS service has three key security features that will be used

1. Assigning service roles that automatically apply security policies based on group and user assignments

2. Identity Access and Management service is native to AWS and can be used to apply multi-factor authentication and other IDAM services.

3. An EC2 key pair can be used to enforce and RSA token for authentication via SSH or PuTTY clients.

# 13. Privacy Concerns

Usernames, credentials, and other sensitive information will be hashed to protect our user’s privacy where possible. AWS services are by default compliant with GDPR. By using AWS as our infrastructure for application hosting we are ensuring we can reach GDPR compliance and assure our customers of their privacy.

# 14. Application Components

Describe the system and the software to which this document applies, including, as applicable:

* Software Name: Carvue Web Application
* Software Version: 1.0
* Release Date: 14 October 2018
* Url: <http://team-flask-app-dev.us-east-2.elasticbeanstalk.com>
* Python Version: 3.7.0
* Flask Version: 1.0
* PostgreSQL Version: 9.6.1
* Jinja Version: 2.10

# 15. What is Carvue?

The below sections describe the overview of the application features, directory overview, the deployment environment, and system performance.

## **15.1. Application Features**

* Intended Use

This web application is designed to support simple CRUD operations for end users to view, select, and purchase vehicles. Users can register with the application, login, and be authenticated to the system. The user can view a dashboard of vehicles. The user can then select a vehicle for purchase and complete a purchase process. The user will then be able to logout. The administrator user will be able to login, add or delete vehicles, and maintain the site.

* Operating Environment

The application is accessible on most major web browsers. It is also mobile friendly by utilizing specific bootstrap and CSS classes to ensure it is rendered correctly on a mobile device. The application will always be available via the AWS infrastructure on Elastic Beanstalk.

* End User Benefits

The main benefit to the end user is that the system is available anywhere they have Internet access. By utilizing a cloud compute infrastructure to host the web application, Carvue provides an always-available car purchasing application. Another key benefit is that the application was built with security as an inherent component. It utilizes unique and robust security functions by default to protect user credentials and privacy.

## **15.2. Directory**

The application directory tree can be viewed via the Git hub repository. The end user will not need to access any particular files and will interact with the application solely through a web browser. The administrator user will also only interact with the application via the web browser.

**Directory Tree**

The application directory tree has several main components. The main logic of the application resides within the application folder. This contains the templates to render web pages, models to handle vehicle and user logic, and session logic. The below images will show the directory tree within the Git repository.

**Directory Use**

If a user or administrator must utilize the directory and operate the application on a local server they may download the files from the Git Hub repository and run the application in the following manner.

1. Upon download, they must configure their local environment to run the necessary dependencies. In this case it is recommended to use the PyCHARM IDE.

2. Open the project folder within the IDE.

3. To run the application run the application.py via the python or python3 command. You should see a similar output as seen below.

## **15.3. Environment Layout**

Environment Description.

The below sections detail the necessary components to operate the system including hardware and software requirements, necessary configurations, and resources. Given the fact that this is a free to use web-based application the requirements are few and it is easy to procure the equipment to access the application. The below recommendations outline what is necessary for hosting and building the application:

* Development Equipment Hardware

i. iMAC 205 27” or newer.

ii. Processor: Core i5 or newer.

iii. RAM: 8GB or more.

iv. Storage: 800GB or more.

* Software

i. Windows 10, Linux, or Mac OSX.

ii. PyCharm IDE.

iii. Amazon Web Services Elastic Beanstalk

iv. Flask

v. PostgreSQL.

vi. Git version control.

vii. Python 3

* Application Hosting

i. Hosted on AWS Elastic Beanstalk.

ii. No physical servers or manual set up required.

* Communication Equipment

i. Required internet connection of 1GB.

ii. Router: TP-Link C7.

# 16. Composition of Application

This section will detail how the composition of the software and the end user experience. There are several expectations that the end user should have when using the system and during operation. The below sub sections provide an overview of performance, how it relates to other systems, and different uses between customers and administrators.

## 16.1. Application Performance

a. Customer

The customer can expect several key characteristics of performance. There will be very low latency as the distributed workload and load balancing of Elastic Beanstalk ensures the application responds quickly. The end user can move between the vehicle pages and dashboards with no latency. The user will be able to list vehicles and select for purchase.

b. Administrator

The admin user can expect access to all vehicles listed from the PostgreSQL database. The admin user type will be able to add, delete, and modify vehicle types through the web browser.

## 16.2 Application Health

Application health can be viewed by the cloud architect via the AWS Management console. Relationship with Other Systems

## **16.3. Version control**

The application is on the master branch of a git hub repository. Any changes to source code will be pushed on the master branch. Development work will be pushed to the git hub repository but on a development branch.

## **16.4. Cloud infrastructure**

The application will then integrate with the version control system. Any changes on the master branch of the git hub repository are automatically reflected in AWS Elastic Beanstalk.

## **16.5. Web Browser**

The code repository that is hosted within Elastic Beanstalk is then available through a web browser via a dedicated URL.

## **16.6. Escalation for Troubleshooting**

If problems arise there are several means of bringing this to the attention of the development and support staff. The below points of contact can be reached for troubleshooting issues:

i. Collin Hicks: chicks39@student.umuc.edu

ii. Jonathan Barnes: jonbar87@gmail.com

iii. Matthew Orahood: [morahood@gmail.com](mailto:morahood@gmail.com)

# 17. System Use and Application Navigation

## **17.1. Application Operation**

The system will be always available via the Elastic Beanstalk instance on AWS. The user simply must navigate to this URL to turn on the application. The application will not require any physical hardware on premises of the application owners. If a custom domain is required please contact the below point of contact for DNS and infrastructure provisioning:

Matthew Orahood: morahood@gmail.com

## **17.2. User Interface**

The user interface is built using the Flask, Jinja2, and dependent frameworks. The styling is accomplished by the integrating with Bootstrap, javascript, and jquery. Each of these three components allows the system to render simple, elegant, and interactive web pages. The table lists these links so the user or administrator can better understand the styling of the application.

|  |  |
| --- | --- |
| Styling Dependencies | |
| **Name** | **Link** |
| Bootstrap | href="<https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css>" |
| Jquery | src="<https://code.jquery.com/jquery-3.3.1.slim.min.js>" |
| Javascript | src="<https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js>" |

## **17.3. Authorization and Authentication**

There are two levels of authorization within the application. A customer user type and Administrator user type. Each has separate levels of authorization within the system. The below table details each user role and specific authorization privileges.

|  |  |  |
| --- | --- | --- |
| Authorization Schema | | |
| **User Type** | **Permissions** | **Constraints** |
| Customer | Read, List, Write | Can only purchase vehicle. |
| Administrator | Read, List, Write, Delete | None |

# 18. Installing the Application

The application does not require a traditional installation process. As this is a cloud-based offering there is no need to download any files or manual installation on a physical device. To access the system, use the following steps.

1. Ensure you have a steady internet connection and are utilizing an up to date web browser.

2. Navigate to<http://team-flask-app-dev.us-east-2.elasticbeanstalk.com/inventory/> .

3. Ensure you are able to navigate through the system and there are no HTTP or request errors.

4. Register for a user account.

5. Login to system and you being looking for vehicles.

# 19. Provisioning the Application

Start the Application. The application, if being hosted by another organization, can be started by the following method.

1. Register with our Cloud Architect for access to the AWS management console.

2. You will be emailed a set of login credentials to the management console.

3. Navigate to the Elastic Beanstalk service.

4. Configure instance to meet your environment needs and apply security controls to meet organizational and industry compliance.

5. Confirm configuration with Cloud Architect.

6. The Cloud Architect will then deploy the Elastic Beanstalk instance and email confirmation.

# 20. Terminating Application Instance

To terminate the application, contact the Cloud Architect. Ensure you have deleted any EC2 keys, credential information, or any other sensitive data. Confirm the assigned URL is no longer accessible via a web browser. The Cloud Architect will send a confirmation email verifying all assets associated with Elastic Beanstalk instance are terminated.

# 21. Application Use and Overview

The Carvue application is easy to use and requires very little in terms of up-front training or instruction. As previously stated, the application is designed to allow a customer to login and view and purchase a vehicle. The below chapter describes the layout, flash messages, coding conventions, and a list of references to other documentation.

## **21.1. System layout**

The basic layout of the web interface portion of the application will be listed below. The application is divided into several sections. There is a home page in which the user can register, login, or navigate to the vehicle dashboard. There is a registration page for first time users. There is also a login page for registered users to gain access to the system. There is a home page dashboard where all vehicles are listed. Each vehicle has its own page to view further details. Finally, there is a purchase page in which the user can enter in their purchase information and purchase the vehicle.

## **21.2. Error Handling Instructions**

* Any errors will be remedied by the development team.
* All malfunctions and emergencies will be handled in the following manner.
  + Immediately exit the application.
  + Report the error to any member of the applicant development team.

## **21.3. Messages**

The application will have the following messages displayed via the flash utility within flask and Jquery.

* Error Messages will be displayed in red.
* Successfully Registered messages will be displayed in green.
* Successfully Logged in messages will be displayed in green.
* Successfully Purchased messages will be displayed in green.

## **21.4. Reference Guide**

* Flask Documentation:<http://flask.pocoo.org/>
* Python Documentation:<https://www.python.org/>
* AWS Documentation:<https://aws.amazon.com/elasticbeanstalk/?sc_channel=PS&sc_campaign=acquisition_USsc_publisher=google&sc_medium=ACQ-P%7CPS-GO%7CBrand%7CDesktop%7CSU%7CMachine%20Learning%7CElastic%20Beanstalk%7CUS%7CEN%7CText&sc_content=elastic_beanstalk_e&sc_detail=aws%20elastic%20beanstalk&sc_category=Machine%20Learning&sc_segment=293647516939&sc_matchtype=e&sc_country=US&s_kwcid=AL!4422!3!293647516939!e!!g!!aws%20elastic%20beanstalk&ef_id=WWdiKQAAAXTHNYkB:20180908172039:s>
* Bootstrap Documentation:<https://getbootstrap.com/>
* PostgreSQL Documentation:<https://www.postgresql.org/>

# 22. Test Plan

## **22.1. Purpose**

This Test Plan describes the testing approach and overall framework driving the testing of our CMSC 495 Project named “Carvue”. This document introduces the following:

## **22.2. Test Strategy**

Rules the test will be based on, including the givens of the project (e.g.: start / end dates, objectives, assumptions) and descriptions of the process to set up a valid test (e.g.: entry / exit criteria, creation of test cases, specific tasks to perform, scheduling, data strategy).

**22.3. Execution Strategy**

Describes how the test will be performed and outlines a process to identify and report defects as well as fix and implement fixes.

**22.4. Test Management:**

Process to handle the logistics of the test and all the events that come up during execution (e.g.: communications, escalation procedures, risk and mitigation, team roster, etc.).

# 23. Project Overview

This project is designed to be a standalone production web application for the CMSC 495 capstone course called “Carvue”. The team designed, developed, and runs a simple web application that is built upon the Flask framework using Python scripts to execute routes and codes. The application utilizes a PostgreSQL database and schema to support the application. It also utilizes a simple CRUD (create, read, update, destroy) functionality and the theme is that of a car buying website. The project requires the role of three contributors. A team lead with full stack development responsibilities, infrastructure and cloud architect, and a database developer/administrator will each contribute to the project.

## **23.1. Audience**

a. Project team members perform tasks specified in this document as well as provide input and recommendations on this document.

b. Project Manager plans for the testing activities in the overall project schedule, reviews the document, tracks the performance of the test according to the task herein specified, approves the document, and is accountable for the results.

c. Quality Assurance ensures that the test plan and deliverables are in line with the design, provides the environment for testing, and follows the procedures related to the fixes of defects.

# 24. Test Strategy

## **24.1. Test Objectives**

The objective of the test is to verify the functionality of Carvue works according to the specifications. The test will verify that the routes, infrastructure, database, and CRUD (create, read, update destroy) functions work as designed.

## **24.2. Test Assumptions**

* Performance testing is not considered for this estimation.
* All the defects will come with a snapshot in JPEG (or similar) format.
* Test environment and preparation activities will be owned by the Dev Team.
* The project will provide test planning, test design, and test execution support.
* Project team has the knowledge and experience necessary, or has received adequate training in the system, the project, and the testing processes.
* There is no environment downtime during tests due to outages or defect fixes.
* The system will be treated as a black box; if the information shows correctly online and, in the reports, it will be assumed the database is working properly.

## **24.3. Test Principles**

* Testing will be focused on meeting the objectives, cost efficiency, and quality.
* There will be common, consistent procedures for all teams supporting testing activities.
* Testing processes will be well defined, yet flexible with the ability to change as needed.
* Testing activities will build upon previous stages to avoid redundancy or duplication of effort.
* Testing environment and data will emulate a production environment as much as possible.
* Testing will be a repeatable, quantifiable, and measurable activity.
* Testing will be divided into distinct phases, each with clearly defined objectives and goals.

## **24.4. Data Approach**

In functional testing, Carvue will contain pre-loaded test data which will be used for testing the application.

## **24.5. Scope and Levels of Testing**

### **24.5.1. Exploratory Testing**

**PURPOSE:** The purpose of this test is to ensure any defects or security flaws are patched before the next level of testing starts

**SCOPE:** All users, to include the admin

**TESTERS:** Testing team

**METHOD:** Exploratory testing is carried out in the application without test scripts or documentation

**TIMING:** At the beginning of the cycle

### **24.5.2. Functional Testing**

**PURPOSE:**  Functional testing will be performed to check the functions of the application. The functional testing is carried out by feeding the input and validates the output from the application.

**SCOPE:** Entire application.

**TESTERS**: Testing team.

**METHOD**: The test will be performed in accordance with the methods defined by the Project Manager or Team Lead

**TIMING**: After Exploratory testing is completed.

### **24.5.3. User Acceptance Testing (UAT)**

**PURPOSE**: The purpose of this test is to focus on validating the business logic. It allows the end-users to complete one final review of the application prior to deployment.

**TESTERS**: The UAT is performed by a/an end user(s).

**METHOD**: Since end users are the most indicated to provide input around business needs and how the system adapts to them, it may happen that the users do some validation not contained in the tests.

**TIMING**: After all other levels of testing (Exploratory and Functional) are done. Only after this test is completed the product can be released to production.

### **24.5.4. Documents**

Placeholder for any test documents.

### **24.5.5. Execution Strategy**

#### **24.5.5.1. Entry and Exit Criteria**

* The entry criteria refer to the desirable conditions in order to start test execution; only the migration of the code and fixes need to be assessed at the end of each cycle.
* The exit criteria are the desirable conditions that need to be met in order to proceed with the implementation.
* Entry and exit criteria are flexible benchmarks. If they are not met, the test team will assess the risk, identify mitigation actions, and provide a recommendation. All this is forwarded to the project manager for a final “go-no go” decision.
* Entry criteria to start the execution phase of the test: the activities listed in the Test Planning section of the schedule are 100% completed.
* Entry criteria to start each cycle: the activities listed in the Test Execution section of the schedule are 100% completed at each cycle.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exit Criteria** | **Test**  **Team** | **Technical Team** | **Notes** |
| 100% of tests executed | Yes | Yes | Application Dev team has conducted all tests |
| 95% pass rate of tests | Yes | Yes | No major issues with application behavior or function. |
| No open Critical or High severity defects | Yes | Yes | No serious defects. |
| 95% of Medium severity defects have been closed | Yes | Yes | This has been concluded. |
| All remaining defects are either cancelled or documented as Change Requests for a future release | Yes | Yes | Will integrate with future change management tool. |
| All expected and actual results are captured and documented | Yes | Yes | Yes |
| All test metrics collected | Yes | Yes | Yes |
| All defects logged | Yes | Yes | Yes |

## **24.6. Test Cycles**

There will be two cycles for functional testing.

1. the objective of the first cycle is to identify any blocking, critical defects, and most of the high defects.
2. The objective of the second cycle is to identify remaining high and medium defects, remove the work-around from the first cycle, and obtain performance results.

UAT test will consist of one cycle.

## **24.7. Validation and Defect Management**

It is expected that the testers execute all tests in each of the cycles described above. It is the responsibility of the tester to open the defects, link them to the corresponding test, assign an initial severity and status, re-test, then close the defect. It is the responsibility of the Defect Manager to review the severity of the defects and facilitate with the technical team the fix and its implementation, communicate with testers when the test can continue or should be halted, request the tester to re-test, and modify status as the defect progresses through the cycle.

Defects found during the Testing will be categorized according to the tools we used. The categories are:

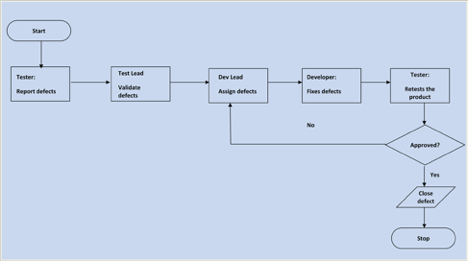
|  |  |  |
| --- | --- | --- |
| **Severity** |  | **Impact** |
| 1 (Critical) |  | This bug is critical enough to crash the system, cause file corruption, or cause potential data loss |
|  |  | It causes an abnormal return to the operating system (crash or a system failure message appears). |
|  |  | It causes the application to hang and requires re-booting the system. |
| 2 (High) |  | It causes a lack of vital program functionality with workaround. |
| 3 (Medium) |  | This bug will degrade the quality of the System. However, there is an intelligent workaround for achieving the desired functionality - for example through another screen. |
|  |  | This bug prevents other areas of the product from being tested. However other areas can be independently tested. |
| 4 (Low) |  | There is an insufficient or unclear error message, which has minimum impact on product use. |
| 5(Cosmetic) |  | There is an insufficient or unclear error message that has no impact on product use. |

## **24.8. Test Metrics**

Test metrics to measure the progress and level of success of the test will be developed and shared with the project manager for approval.

## **24.9. Defect tracking & Reporting**

The flowchart below depicts Defect Tracking Process:



## **24.10. TEST MANAGEMENT PROCESS**

### **24.10.1. Test Management Tool**

AWS Inspector will be used for testing. All testing artifacts such as test cases and test results will be tracked and updated in Microsoft Excel.

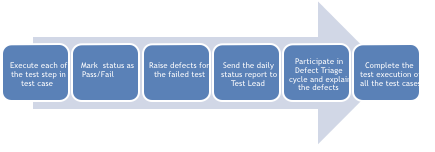
### **24.10.2. Test Design Process**

|  |
| --- |
|  |

* The tester will understand each requirement and prepare corresponding test case(s) to ensure all requirements are covered.

### **24.10.3. Test Execution Process**

|  |
| --- |
|  |



|  |
| --- |
| all the test cases |

## **24.11. Test Environments**

Test cases will be run on the application hosted in AWS. A Windows, Apple, or \*nix environment with the latest version of IE, Firefox, or Chrome should be available to each tester.

## **24.12. Approvals**

The Names and titles of all persons who must approve this plan:

|  |
| --- |
| **Name:Collin Hicks** |
| **Role:Security Tester** |
| **Date:10/10/2018** |
| **Signature:Collin Hicks** |

|  |
| --- |
| **Name:Matthew Orahood** |
| **Role:Infrastructure and Cloud Architect** |
| **Date:10/10/2018** |
| **Signature:Matthew Orahood** |

# 25. Design Overview

Carvue aims to be the premier web-based application that simulates the car buying process in the modern compute age. The product is designed to be used in an academic setting and is designed to be tested as an academic project. However, the application will simulate a production deployment on AWS. The main purpose of this section is to provide an overview of the drivers within the project that led to its creation.

## **25.1. Background Information**

The Carvue application was conceived and created to meet the specifications of a fully completed project for the capstone CMSC 495 class. The idea was presented by our Cloud Architect, Mathew Orahood, to create a clone of the Carvana web application and use the Flask framework to accomplish this. To show the versatility of a modern computing environment we decided upon using the Elastic Beanstalk service in AWS to host the application. Additionally, we decided to use the most popular version of version control, Git, to act as the method of creating a code repository. The course is designed to model a software development process used in a full production environment. The application, as a result, needed to be developed using an SDLC process in tandem with a SCRUM process. The desired end state is to have a fully functioning web application that can:

* Complete CRUD operations.
* Define users and separates them based on authorization and authentication.
* Show an understanding of the cloud tools offered by AWS.
* How to properly apply version control.
* How to secure and test a web application for vulnerabilities.

## **25.2. Application Migration**

The existing system is built upon the AWS services architecture. Our development team highly recommends using AWS Elastic Beanstalk to host the application. If there is a requirement to host on a different service all necessary directories will be provided via the Github repository.

## **25.3. Current Application Hosting**

The application is currently hosted via AWS Elastic Beanstalk at: http://team-flask-app-dev.us-east-2.elasticbeanstalk.com

## **25.4. Technology Forecast Integrations**

The following technology integrations are being considered for future possibility depending on requirements and new compliance directives:

1. Multi Factor Authentication via Google Authenticator.

2. Azure cloud deployment for redundancy.

3. Okta single sign on integrations.

## **25.5. Constraints**

1. Schedule Constraints:

a. Full functionality must be deployed no later than 7 October.

b. Security testing must be completed no later than 9 October.

c. AWS provisioning must be completed no later than 10 October.

2. Security Constraints.

a. Must meet NIST 800-53 compliance.

b. Must meet PCI DSS compliance once available for public use.

c. Must integrate a logging solution if fully public.

3. Design Considerations

This project presented several considerations in terms of choosing between flexibility, security, and a combination of other factors. As a development team, we went through a process of determining which elements of the Flask framework, the python language, and the other dependencies would affect the end user experience. This also includes the security implications and how security controls affect the user experience. The below subsections detail the design considerations of the major components that make up the Carvue application.

# 26. Flask

The flask framework is highly flexible and allows for an easy to use means of creating web applications. Its main benefit, and its reason for selection, is that it is easy to integrate security controls that are native to the framework. As an example, there are built in extensions for tokenization and authentication. Even more importantly these integrations do not create a significant difficulty from the end user perspective of accessing the systems.

# 27. Stripe Integration for Payments

Another tradeoff that was central to development was how to integrate a payment system into the Carvue application. After consideration of other payment options, like square, the issue of ease of use led the development team to choose Stripe. The reason for this selection is that it has a rich set of APIs that meant the integration into the Flask framework

# 28. AWS for Reliability

Another consideration was the ability to ensure the application could be hosted without interruption and would always be available. In addition, there needed to be a centralized method to track and provision users who would develop and administer production assets. To meet these demands, we selected Amazon Web Services as the means to host the application with their Elastic Beanstalk product. The ease of use, redundancy, and resource management inherent within AWS allows the development team to quickly update the Carvue application to meet new requirements and do so in a secure fashion.

# 29. User Characteristics

This web application is designed to be an example of a commercially available application that would be used by an average consumer. However, it is more aligned to be evaluated and used in an academic setting. Therefore, the average user will be a student completing the CMSC 495 capstone course as well as the instructor. This application is designed to be easy to use to show the flexibility of the Flask Framework with Python and cloud computing.

# 30. User Challenge

The major challenge facing the user base is the use of an application within the academic setting that accurately mirrors a full-fledged enterprise level production application.

# 31. User Requirements and Needs

The following list will detail the requirements as needed for users of the Carvue application.

1. Create a sample application and associated design documentation.

2. Create a CRUD application that allows users to view, select, and purchase vehicles.

3. Integrate a payment system such as Stripe to simulate real purchases.

4. Create a version control repository on Git to make the application files accessible and open source.

5. Have a public elastic beanstalk URL that allows users to access the application.

6. Ensure there are two user types, customer and administrator.

# 32. System Architecture

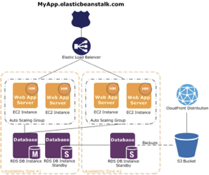
## **32.1. Hardware Architecture**

The hardware architecture for Carvue is unique in that it is a fully cloud delivered solution. As a result, the hardware footprint is quite small. The main hardware that is used are desktop devices for development. The rest is commodity hardware provided automatically by AWS. The below diagram is an architectural design of the AWS deployment for the Carvue application. As you can see the application is in a distributed architecture that ensures the application is always available. To handle spikes in traffic and requests the EBS instance also utilizes load balancing to ensure reduced latency. As AWS uses commodity hardware and we only utilize portions of each it is difficult to create a fully annotated list of hardware specifications. However, we can specify the workstation used for development. The below table will detail the workstation used for development and pushing changes to the Github repository.

|  |  |
| --- | --- |
| Name | Specification |
| Endpoint | Apple iMac 2015 27 inch |
| Operating System | Apple OSX Version 10.12 |
| Memory | 16GB Configurable Onboard Memory |
| Storage | 1 TB Fusion Drive |
| Processor | 3.1GHz quad Intel core i5 |
| Ports | 4 USB, 2 Thunderbolt, Gigabit Ethernet, SDXC Card Slot. |
| Number of Endpoints Used | 2 |

## **32.2. Software Architecture**

The software architecture is built upon two main components. The Flask framework is the vehicle in which the actual development of the application occurs. Carvue is written in Python to handle application logic and PostgreSQL for database management and storage. Within Flask there are a plethora of extensions and libraries that can be used to add modules to the application. In conjunction with Flask the PyCharm IDE is used to write the Python code and develop the application. Version control of the software is handled by Git with a repository available online to add and commit file changes. The below table will detail the overall system organization of the software package.



As you can see, the application is in a distributed architecture that ensures the application is always available. To handle spikes in traffic and requests the EBS instance also utilizes load balancing to ensure reduced latency. As AWS uses commodity hardware and we only utilize portions of each it is difficult to create a fully annotated list of hardware specifications. However, we can specify the workstation used for development. The below table will detail the workstation used for development and pushing changes to the Github repository.

|  |  |
| --- | --- |
| Name | Specification |
| Endpoint | Apple iMac 2015 27 inch |
| Operating System | Apple OSX Version 10.12 |
| Memory | 16GB Configurable Onboard Memory |
| Storage | 1 TB Fusion Drive |
| Processor | 3.1GHz quad Intel core i5 |
| Ports | 4 USB, 2 Thunderbolt, Gigabit Ethernet, SDXC Card Slot. |
| Number of Endpoints Used | 2 |

|  |  |
| --- | --- |
| Name | Specification |
| Flask Version.1.10 | atomicwrites==1.2.1  attrs==18.2.0  certifi==2018.8.24  chardet==3.0.4  click==6.7  Flask==1.0.2  Flask-SQLAlchemy==2.3.2  Flask-WTF==0.14.2  idna==2.7  itsdangerous==0.24  Jinja2==2.10  MarkupSafe==1.0  more-itertools==4.3.0  pluggy==0.7.1  psycopg2==2.7.5  py==1.6.0  pytest==3.8.0  requests==2.19.1  six==1.11.0  SQLAlchemy==1.2.11  stripe==2.8.0  urllib3==1.23  Werkzeug==0.14.1  WTForms==2.2.1 |
| Python | Version 3.6 |
| PyCharm | Version 2.2 |
| AWS Elastic Beanstalk | Version 2.6.1 |
| PostgreSQL | Version 10.5 |

This table is only one piece of understanding the overall software architecture. The Carvue application can be further detailed and understood in how data flows between systems, services, and how version control affects this process. The below section will detail step by step the overall architecture of Carvue.

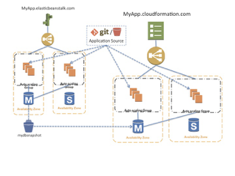
1. Code is written and saved within the PyCharm Development IDE environment.

2. Git version control is initialized and the developer copies the origin of the repository into the git file.

3. All changes are added, committed with comment, and then pushed to the master branch of the Git repository.

4. As soon as any change is pushed to Git, AWS Elastic Beanstalk immediately reflects these changes and provisions the necessary resources to account for these updates.

5. The below screenshot is a sample of this process



# 33. Network Architecture

The network architecture relies upon a simple yet effective use of WAN networks and AWS infrastructure components. There are two main components to the network architecture. The first is the local development environment, which is simply a standard LAN configuration. The second is the AWS infrastructure which actually hosts and makes the application available for public consumption.

1. LAN Architecture

2. The LAN architecture used for development is quite simple. It consists of a client, in this case our iMac 2015, and a router that then accesses the Git repository URL for changes to be pushed to. The below table documents the components of the LAN network.

|  |  |
| --- | --- |
| LAN Component | Function |
| Client: iMac 2015 | Client in LAN architecture. Houses PyCharm IDE and used to develop application before pushing to Git repository. |
| Router: | Router that forwards packets to Edge Gateway. |
| ISP Provider: Verizon | Verizon acts as the ISP provider for this project. |

# 34. AWS Architecture

The AWS network architecture is much different than a traditional LAN or WAN architecture. The basic tenant of AWS is that services should be scalable, available, and resilient. As a result, many of the resources are distributed and can be shared or isolated logically and geographically. This is accomplished in several ways. The first component of the architecture is that AWS provides regions in which resources are located. These are geographically isolated and within them contain another subdivision of availability zone to further provide scalability. The below diagram provides a topology of the AWS infrastructure.



# 35. Data Design

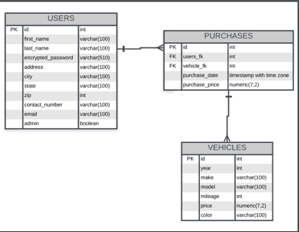
This section will detail the use and layout of the Carvue database schema and its associated files and components. As stated early in this document the Carvue application is built using the PostreSQL database to act as the database and data repository.

## **35.1. PostgreSQL Schema**

PostgreSQL is an object relational database management system that is extremely extensible and allows for highly scalable support to web applications. The PostgreSQL database will hold all the defined objects that will be used within the Carvue application. The below diagrams will provide more context and detail regarding the schema and logical modeling of the database.

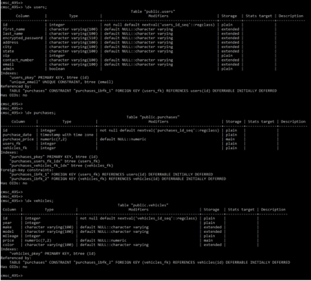
## **35.2. ERM Logic Model**

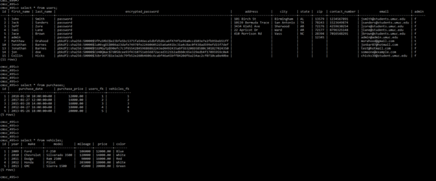
The below ERM diagram shows the relationships between objects within the Postgre database. The logic of the application is quite simple. There are users who can make many purchases related to many vehicles. The below diagram will detail this below:



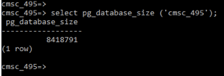
## **35.3. Database Schema**

The database schema was made using PostgresSQL syntax. The below screenshots detail the database schema that will be used in the Carvue application.



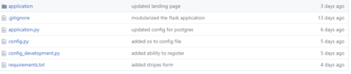


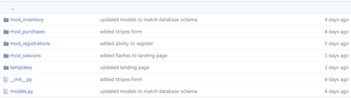
The schema also can reveal further details regarding the Postgre database. For example, we can query the database to see its total schema size and ensure our AWS environment can support that amount of data. The below screenshot shows the output of querying for this information.



## **35.4. Non-Database Management System Files**

The non-database files have been previously discussed in this document as those files that are contained within our version control system through Git. These files are used to create the application, are developed in PyCharm, and automatically pushed to Elastic Beanstalk via Git. The below tables and screenshots will further describe the components of the file directory. The below screenshot will the overall directory taken from the Git repository.





The above screenshots show the basic overview of the directory tree. The below table will detail each sub directory, its sub components, and use within the Carvue application.

|  |  |
| --- | --- |
| Directory & Files | System Use or Function |
| application | Contains major application components, models, views, and controllers. |
| .gitignore | Contains EBS files. |
| application.py | Python file that executes and runs application. |
| config.py | Configures database, database connections, and secret key. |
| config\_development.py | Same as config.py except for development environment before being pushed to master branch and EBS. |
| requirements.txt | Contains and lists all libraries, dependencies, and integrations used within the flask framework. |
| mod\_inventory | Contains the MVC files for vehicles. |
| mod\_purchases | Contains the MVC files for purchases. |
| mod\_registrations | Contains the MVC files for registration process. |
| mod\_sessions | Contains the MVC files for the session management. |
| Init.py | Imports all mod sub directories and pushes to main application. |
| models.py | Imports database to models and add abstraction. |

# 36. Detailed Design

This chapter describes the proposed design in detail. Provide the necessary information for the development team to integrate the hardware components, write the software code, so that the hardware and software components will provide a functional product.

Every design item should map back to the Functional Requirements Document. These should be captured in the Requirements Traceability Matrix.

# 37. Hardware Detailed Design

The Carvue project takes advantage of various Infrastructure as a Service (IAAS) and Platform as a Service (PAAS) offerings made by AWS. IAAS affords us the opportunity to maintain only the operating system and the application running on the virtualized hardware maintained by Amazon Web Services (AWS). Conversely, PAAS only requires us to maintain the application and data without worrying about operating system updates.

Components [application and database] of our application are separated across two different pieces of hardware. The Flask application leverages AWS ElasticBeanstalk which is an IAAS offering at Amazon and runs on a AWS EC2 t2.micro general purpose machine. This machine uses 1 virtual cpu (VCPU) and 1Gib of memory. AWS ElasticBeanstalk provisions a load balancer for us and provides an easy to use interface for defining elasticity rules. Elasticity, as defined by AWS, allows our infrastructure to provision and decommission additional EC2 instances based on the current demand.

It is important to note that the t2.micro is a burstable performance instance. This means that our machine is allocated a certain amount of CPU credits when first initialized. We are guaranteed a certain level of baseline performance on our instance but our CPU credits allow us to extend beyond that baseline during periods of high activity. Periods with lower than normal activity allow us to collect and save CPU credits for times when we do need a burst in performance.

Carvue’s database also runs on an EC2 t2.micro however, it is provisioned through the AWS Relational Database Service (RDS). AWS RDS is a PAAS offering specifically catered to various different types of relational database resources (e.g. MySQL, AWS Aurora, Postgresql, MariaDB, Oracle, SQLServer). AWS handles the patching, provisioning and backups of our relational database.

# 38. Software Detailed Design

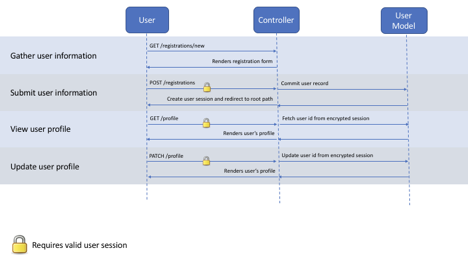
The Flask application’s core functionality is broken up into several distinct modules:

* mod\_inventory
* mod\_purchases
* mod\_registrations
* mod\_sessions

## **38.1. Registrations Module (mod\_registrations)**

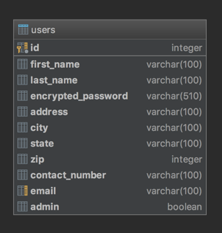
Registration allows users of our application to create an account so that they can enjoy a more customizable experience. This module will be responsible for providing a functionality for user account creation and modification.

### **38.1.1. Processing**

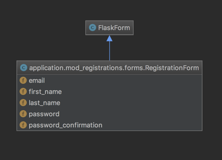


### **38.1.2. Local data structures**

When a user creates an account, a record is saved to the database representing the user account. Consider the following image that represents the fields included in a user record:



Flask utilizes a dependency to help build more secure web forms. Web forms can be represented as Python objects and can be subjected to strict validation requirements. Consider the following image that represents the user registration form:



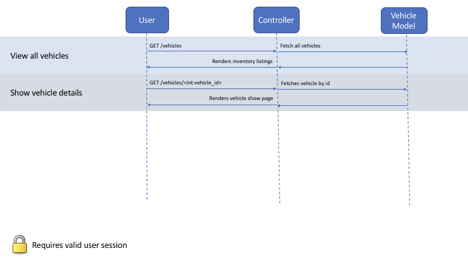
The RegistrationForm object validates the user’s input and requires the following fields:

* Email must be provided
* First name must be provided
* Last name must be provided
* Password must be provided
* Password confirmation must be provided and must be equal to the password

## **38.2. Inventory Module (mod\_inventory)**

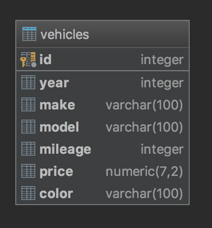
The inventory module is responsible for providing functionality that helps users and administrators view, modify and otherwise interact with vehicles in our inventory.

### **38.2.1. Processing**



### **38.2.2. Local data structures**

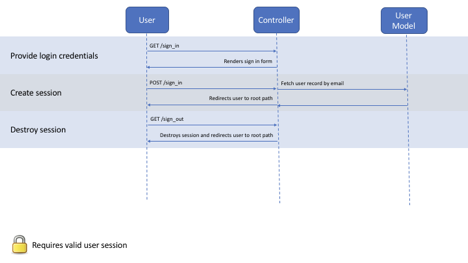
Our application serves the automotive community and the Vehicle object represents an inventory item. Consider the following image which represents a vehicle record within the vehicles table:



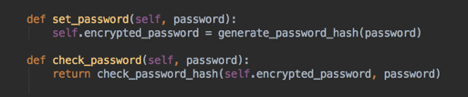
## **38.3. Sessions Module (mod\_sessions)**

The sessions module is responsible providing functionality for users to securely sign in and sign out of our application. The Flask framework provides a framework for generating encrypted session cookies that can store arbitrary session information. For the purposes of this course, our sessions simply store the user’s unique id.

### **38.3.1. Processing**

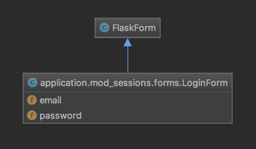


User passwords are encrypted at rest which means we cannot simply compare the value in the database with what is provided in the login form without first securely hashing the provided password value and then comparing the values. The werkzeug.security dependency provides methods for us to do such a computation. Consider the following code snippets as an example of how passwords are encrypted and checked:



### **38.3.2. Local data structures**

Using the wtforms dependency, a LoginForm is a Python representation of our log in form. Consider the following example:



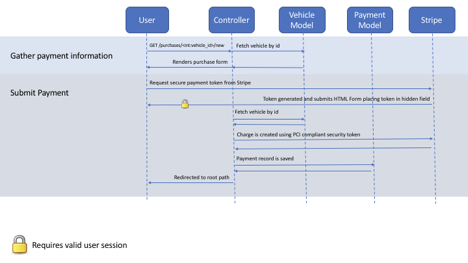
The LoginForm class requires the following validations:

* Email must be provided
* Password must be provided

## **38.4. Purchases Module (mod\_purchases)**

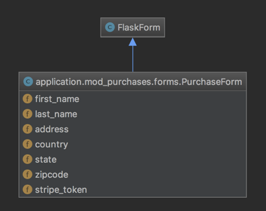
The purchases module is responsible for providing a mechanism to allow users to purchase a vehicle. Carvue uses Stripe ([https://stripe.com](https://stripe.com/)) for secure payment processing. No sensitive payment information is ever passed through the Flask application.

### **38.4.1. Processing**



### **38.4.2. Local data structures**

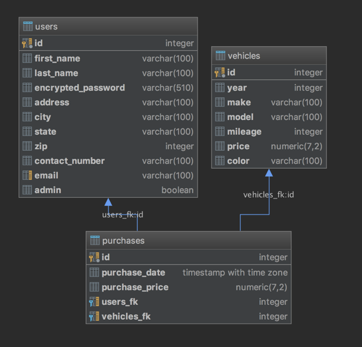
The purchase form is represented as a Python object called PurchaseForm, consider the following image:



The PurchaseForm object executes the following validations:

* First name must be included
* Last name must be included
* Address must be included
* Country must be included
* State must be included
* Zipcode must be included
* Stripe token must be included

The purchase model has several foreign keys including the user who purchased the vehicle and the vehicle being purchased. Consider the following image as a representation of this relationship.



# 39. Communications Detailed Design

The overall networking of the application is fairly simple since it includes only two primary components:

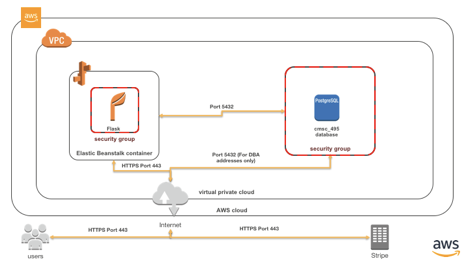
1. AWS Relational Database Service (RDS) (Postgresql)

2. AWS ElasticBeanstalk Application Servers (Python Flask Application)

Infrastructure is hosted in AWS and is nested inside of what is known as an AWS Virtual Private Cloud (VPC). VPC allows you to segregate resources into private networks for added security. Additionally, VPC allows you to assign network ingress/egress rules. These network ingress/egress rules are stateless which makes them stricter than stateful rules. For example, if you wanted to transmit and receive over port 80 you have to create an ‘allow’ rule for both ingress and egress traffic. AWS VPC will implicitly deny anything that is not included in your ruleset. It is also important to note that AWS Elastic Cloud Compute (EC2) instances also have their mechanism to protect their traffic. Security groups are defined at the resource level and are considered stateful meaning that it will automatically allow egress on port 80 if you allow ingress traffic on the same port. This layered approach to security allows administrators and developers to fine tune their network security rules to best suit their application.

# 40. Communications Architecture Overview

Consider the following network topology which illustrates the communication strategy between components of the Carvue application.



# 41. External Interface Design

The Carvue application only interfaces with one system that is not included within the scope of the system under development.

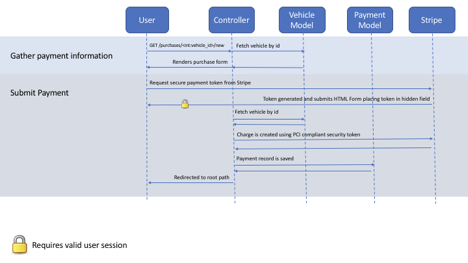
## **41.1. Stripe**

Stripe handles all payments for the platform and relieves Carvue developers of the burden of remaining PCI-certified. As stated on Stripe’s website [<https://stripe.com/docs/security/stripe>]:

*“Stripe has been audited by a PCI-certified auditor and is certified to PCI Service Provider Level 1. This is the most stringent level of certification available in the payments industry. To accomplish this, we make use of best-in-class security tools and practices to maintain a high level of security at Stripe.”*

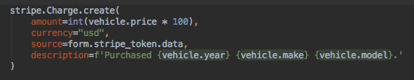
Stripe was specifically designed to relieve the burden of PCI compliance for their customers. There are multiple ways to integrate Stripe into a system however Carvue will take advantage of the JavaScript and Python libraries created by Stripe.

To remain compliant, the Carvue system must not have any credit card details being sent across its systems. Instead, we embed Stripes’ credit card form fields inside the Carvue’s purchase form using the Stripe JavaScript library. When a user submits the form, the default send behavior is prevented and the credit card information is sent directly to Stripe by an AJAX request. Stripe will then generate a secure token for this credit card which can be safely passed through the Carvue system without concern for PCI compliance. When the AJAX returns the token, it places the token in a hidden field and submits the Carvue purchase form. Once the Flask application has received the secure token, the charge can be completed by submitting a charge request to Stripe using the Stripe Python library. The image below depicts this communication flow:



## **41.2. Stripe Detailed Design**

Using the Stripe Python and JavaScript libraries will automatically conform the Flask application with the appropriate messaging formats. It is important to note that Stripe conducts transactions in the currency’s lowest denomination. For example, if the currency is the United States Dollar (USD) then our charge amount needs to be in cents. Consider the following image which shows how simple it is to make a charge using the Stripe Python library:



## **41.3. Human-Machine Interface**

Carvue is only accessible through a web browser. Using HTML, JavaScript and CSS, developers will be responsible for creating web accessible interfaces for all workflows.

## **41.4. Interface Design Rules**

The web application should be accessible and usable by mobile devices. Carvue will use Bootstrap ([https://getbootstrap.com](https://getbootstrap.com/)) as a framework for building a responsive design.

## **41.5. Inputs**

The primary method of user input will be the following web forms:

* User registration
* Profile modification (valid sessions only)
* Session creation/deletion
* Purchasing (valid sessions only)
* Inventory creation (administrators only)
* Inventory modification (administrators only)

## **41.6. Outputs**

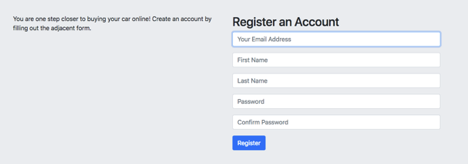
The outputs are unique to the resource being accessed and can be further understood by reviewing the screenshots provided subsections 7.4.1-7.4.5.

## **41.7. Navigation Hierarchy**

Carvue’s navigation remains static throughout while navigating the entire website. Specific calls to action relevant to the current resource will be located within the body of the HTML document. The navigation is included at the top of every page in our application. Consider the following image:

## **41.8. User Registration**

Users will be able to register when visiting the ‘/registrations/new’ endpoint by filling out the following form:



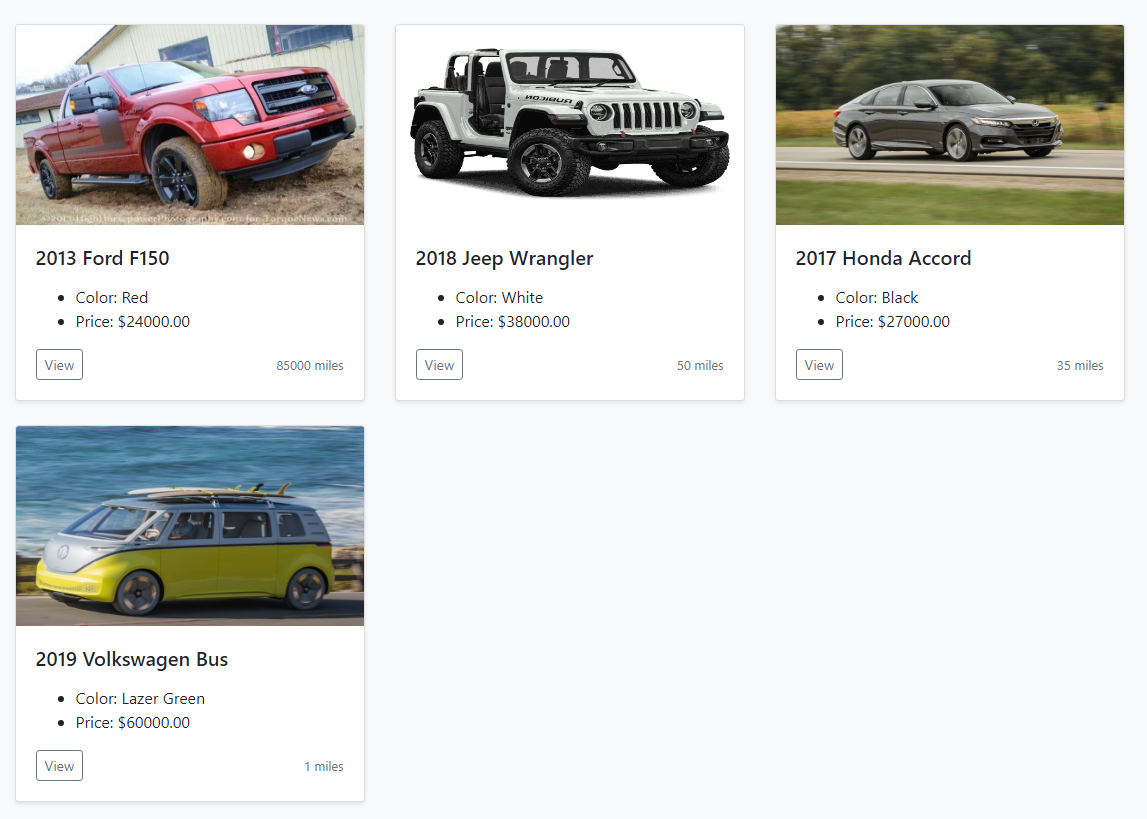
## **41.9. Session Creation**

Users will be able to sign in to their existing account when visiting the ‘/sign\_in’ endpoint by filling out the following form:



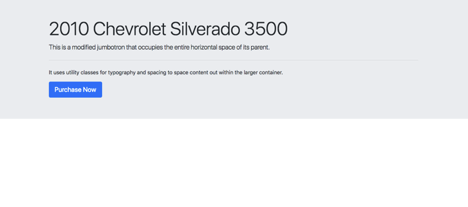
## **41.10. Inventory Listing**

Users will be able to browse a listing of vehicles when visiting the ‘/vehicles’ endpoint, consider the following prototype image:



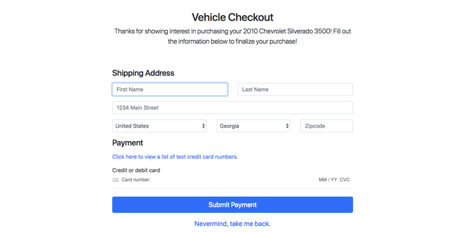
## **41.11. Vehicle Details**

Users can find more details about a specific vehicle when visiting the ‘/vehicles/<int:vehicle\_id>’ endpoint. Consider the following prototype image:



## **41.12. Vehicle Purchase**

Users can purchase a vehicle when visiting the ‘/purchases/<int:vehicle\_id>/new’ endpoint by filling out the following form:



# 42. System Integrity Controls

The Carvue application was designed with security inherent within itself. There are several key security features that will be described and detailed. These areas will include the processed for the creation, update, and deletion of data. This section will then detail proposed security controls that will be tested and verified through a variety of manual and automated processes.

# 43. Verification for CRUD operations

The system will allow both Customer user types and Administrator user types to complete CRUD operations. However, only the Administrator user type will be able to update, delete, or create new vehicles. Customer user types will only be able to view and update certain vehicle data. To verify these permissions are functioning properly, the following tests will be conducted and verified.

|  |  |  |
| --- | --- | --- |
| Test Case | Expected Outcome | Verification |
| Customer User Views Vehicle | Customer user type is able to select vehicle from dashboard and be routed to its individual page. | View button on vehicle container on dashboard routes user to vehicle page. |
| Admin User Adds Vehicle | Admin user is able to add vehicle that is then displayed on both dashboard and individual vehicle page. | Query of Database shows migration of new data into all necessary fields. |
| Admin User Deletes Vehicle | Admin user is able to delete vehicle. | Query of Database shows migration of deleted data from all necessary fields. |
| Customer User Is Unable to Add Vehicle | Customer does not see an add vehicle button on the dashboard. | Verify by using customer account to see if add vehicle function is displayed. |
| Customer User is Unable to Delete Vehicle. | Customer does not see delete vehicle button on the dashboard. | Verify by using customer account to see if delete vehicle function is displayed. |

# 44. Security Control Tests

The below table will detail the major security tests that will be conducted on the Carvue application. It should be noted that these tests will be conducted not on the production site but on a local development site. The reason for this is twofold; one, we do not have permissions to conduct penetration tests on AWS resources. Second, a test on the production site could affect access to other team members and users of the application. The testing will be conducted using the AWS Inspector tool. There are three main security tests that will need to be conducted. There will be tests to determine if SQL injection attacks are possible, if directory traversal is possible, and to see if the application is vulnerable to Cross Site Scripting attacks.

|  |  |  |
| --- | --- | --- |
| Security Vulnerability | Expected Outcome | Possible Remediation |
| SQL Injection Attack | ZAP Proxy attack may uncover SQL Injection vulnerability stemming from incorrectly formatted prepared statements | Ensure prepared statements are used in Postgre database for all tables. |
| Directory Browsing | ZAP Proxy attack may uncover directory browsing vulnerability stemming from a lack of index files. | Convert directory to .httaccess or ensure index files are located in all directories and sub directories. |
| Cross Site Scripting Attack | ZAP Proxy attack may uncover the ability to conduct a XSS attack due to the lack of character stripping | Ensure Jinja 2 is properly formatted so all escape and special character values associated with xss attacks. |

# 45. Conclusion

This project presented a unique opportunity to demonstrate proficiency in project management, python development, and the ability to use the Flask framework. Our group was able to rapidly adapt to a new framework, programming language, and infrastructure to develop an easy to use web application. The level of effort put forth by the group ensured that this project was successful, met our expectations, and was on time for delivery.

In terms of design strength, the ability of the Flask framework to integrate functionality, security, and design into one view greatly enhanced our application. The easy to import libraries for Flask meant that we were able to develop and implement changes on the fly without serious code revisions. This extensibility was also found in the use of AWS Elastic Beanstalk. This cloud service provided a full ecosystem and infrastructure to host our web application.

The true benefit was that the provisioning of servers, resources, and backend infrastructure was automatic and all that was required on our part was to upload our python code directory to AWS. We were also able to integrate this with our Github repository which meant as soon as a change was pushed to the repository, it was automatically reflected on the web application. This provided a strong method to rapidly push development changes to the web application.

Ultimately, there were some limitations in terms of project development and deployment. One issue was we did not utilize a single container or service for development. This meant that development occurred on local machines and if there was an issue in initializing the git repo changes could not be pushed easily to the application. To address this, I would recommend further development should occur in a container service, such as Cloud9 or Nitro, that would provide a common area for development to occur.

This project presented both a great challenge and opportunity to showcase the ability to integrate security, development, and computer science concepts. Learning a new programing language, framework, and infrastructure in a matter of 8 weeks inspired us to try new ideas and methods of solving problems. This course offered the ability to think critically and develop technology in a manner that taught valuable concepts.