MYES Bus Ticketing System

Revision History

Date	Author(s)	Description	Version
05/05/2021	Mustafa Ilıkkan Serdar Mumcu	Initial Version	v1.0
09/05/2021	Mustafa Ilıkkan Serdar Mumcu	Addition of accepted changes from Iteration2 review.	v1.1
23/05/2021	Mustafa Ilıkkan Serdar Mumcu	DB diagram in DataView section is updated	v1.2
30/05/2021	Mustafa Ilıkkan Serdar Mumcu	Updated architectural goals and philosophy, assumptions and dependencies sections, architectural memos, and deployment view sections.	v1.3
06/06/2021	Mustafa Ilıkkan	Addition of accepted changes from Iteration3 review.	v1.4
20/06/2021	Mustafa Ilıkkan	Updated Use Case View and finalized the architecture notebook after conducting a full review.	v1.5

Architecture Notebook

1. Purpose

The purpose of this document is to describe the architectural goals of the MYES Bus Ticketing System. This document lists the assumptions and dependencies as well as the decisions, constraints and justifications that relate to the architecture. Decisions, constraints and justifications that relate to the architecturally significant requirements are noted as technical memos. Architectural mechanisms used in the system and the different architectural views are presented in the relevant sections.

This document aims to give an overview of the architecture for all the new team members. Relevant sections are updated during each iteration. Updates are reflected in the change logs section.

2. Architectural goals and philosophy

1. Modularity is an important concern for the MYES Bus Ticketing System. The software should be built as having low coupling between components. System should be modifiable, extensible and open for future changes. In summary, architecture should support extensibility and low coupling.

2. Another important concern is portability. We will not maintain our physical servers. Instead, we will use cloud resources from Amazon Web Services (AWS). We will deploy the application on an Amazon Elastic Compute Cloud (EC2) virtual machine. Although we are deploying to AWS currently, we do not want to be tied to a particular cloud vendor. Using the infrastructure-as-code approach and Docker Swarm tool, we can move the software to another cloud vendor (such as Azure, Google Cloud, or Digital Ocean) without problem.

3. We aim for a rapid and incremental development style. We will use the Django web framework to achieve this goal. When we evaluated web application framework options available in the market, Django seemed the most viable option due to its support for general usage web application development as well as its big ecosystem and detailed documentation.

4. We also value the scalability of the system. We will use Docker to containerize the application for easy and fast deployment. We will use Docker Swarm to achieve scalability and orchestration. Currently we deploy the software on a single machine for demonstration purposes. But we have scalability in mind from the beginning and we aim to scale rapidly according to the load using the Docker Swarm tool.

5. After analyzing our problem domain, we came up with using a relational database management system because it was more suitable for a relational data model like ours. We are using SQLite relational database management system for the first release but we are implementing the database as a separate layer. This practice will give us the opportunity to easily switch to another database management system provided that it is a relational one.

6. Reliability, security, performance, usability, and supportability qualities will be mostly provided by the cloud platform so that we can concentrate primarily on the development of functional requirements and application logic more efficiently.

7. We will use Docker Compose to provide a standard and sterilized development environment across development team members, and to overcome problems caused by the differences between development and deployment machines.

8. We will use the “git” tool for version control and GitHub as the git provider for collaboration during development activity.

3. Assumptions and dependencies

1. We assume that MYES Bus Ticketing System will be running on the cloud as a SaaS application.
2. The team is composed of experienced developers and testers. We assume that team members are familiar with all tools and technologies that will be used throughout the project.
3. Technologies used for development are open source and flexible technologies.
4. Cloud infrastructure on which we will deploy our application is assumed to be highly available.
5. The user should have JavaScript enabled because some user interface components are rendered by JavaScript on the browser.

4. Architecturally significant requirements

Architecturally significant requirements are the non-functional requirements that are specified in the System-Wide Requirements Specification. These consist of the below system qualities.

1. Usability requirements (section 3.1 in system-wide reqs.spec.)
2. Reliability (section 3.2 in system-wide reqs.spec.)
3. Performance (section 3.3 in system-wide reqs.spec.)
4. Supportability (section 3.4 in system-wide reqs.spec.)
5. Security (section 3.5 in system-wide reqs.spec.)

5. Decisions, constraints, and justifications

Technical Memo-1	
Issue	System architecture decision
Solution	We will use layered (3-tier) architecture
Motivation/Justification	• To organize the system into discrete layers • To provide collaboration and coupling from higher to lower layers • To improve cohesion in coarse grained layers
Alternatives considered	None

Technical Memo-2	
Issue	Selection of programming language and web development framework
Solution	Python and Django web framework will be used for development *(Supportability Req. 3.4.3)*

Motivation/Justification	• Django provides rapid application development, code-first approach development (database generation from code), automatic admin interface generation, object-relational mapping and database migration support, and easy form template generation. • It also comes with automatic authorization and authentication mechanisms and security middleware support • Design philosophies and features of the Django web framework on which our application's architecture is based are Loose coupling, Less code, Quick development, Do not repeat yourself (DRY), Explicit is better than implicit, Consistency at all levels, Include all relevant domain logic, SQL efficiency, Powerful syntax, Loose coupling, Separate logic from presentation (template system), Be decoupled from HTML, Safety and security, Extensibility.
Alternatives considered	Java (Spring MVC framework), Ruby (Ruby on Rails)

Technical Memo-3	
Issue	Selection of technology for the user interface layer
Solution	• HTML, CSS (Bootstrap library), and javascript (JQuery library) will be used for the user interface. *(Supportability Req. 3.4.5)* • Django's template language will be used to create necessary files for the rendering of the user interface.
Factors	• We will support only Chrome (version 89 or later) for this release of the system. • We will not support mobile browsers or implement a mobile native application in this release. • For the table views on the user interface, we will use the Bootstrap Datatable component.
Motivation/Justification	• Browser understands HTML, CSS and JavaScript. • Bootstrap CSS framework helps build UI and apply styling without diving into complex CSS selectors. • The JQuery library helps DOM (Document Object Model) manipulation and handling interactivity at user interface level easier compared to plain JavaScript. • Django templates are generic files to allow rendering of different dynamic views depending on the data coming from the server side.
Alternatives considered	Plain JavaScript, plain CSS (Cascading Style Sheet) language

Technical Memo-4	
Issue	Selection of database management system

Solution	• A relational database management system will be used. • For the initial release, we will use the SQLite database, but in the future it can be changed to other relational database management systems such as MySQL, PostgreSQL, or Oracle. (it is possible to switch the database by changing some configuration inside Django framework)
Motivation/Justification	• After analyzing our problem domain, we came up with using a relational database management system because it was more suitable for a relational data model like ours.
Alternatives considered	NoSQL databases

Technical Memo-5	
Issue	Decision of the deployment model
Solution	• Software will be deployed on cloud in a Docker container *(Supportability Req. 3.1.2, 3.4.1 and 3.4.2)*
Motivation/Justification	• Reliability, security, performance and usability qualities will be mostly supported by the cloud platform so that we can concentrate primarily on the development of functional requirements and application logic more efficiently • Docker containerization will provide easy and fast deployment. • The docker swarm tool will be used for container orchestration. Orchestration will provide scalability, load balancing, easy roll back, and desired state reconciliation. • The docker machine tool will be used for cloud deployment. It helps provision AWS EC2 instances from the command line. • The docker hub platform will be used as the docker image repository. • Although AWS is chosen for now, we can easily deploy to another cloud vendor using Docker Swarm.
Alternatives considered	On-premises deployment, deployment on PaaS services (Heroku), deployment on other cloud vendors (Microsoft Azure or Google Cloud)

Technical Memo-6	
Issue	Setting up the software development environment

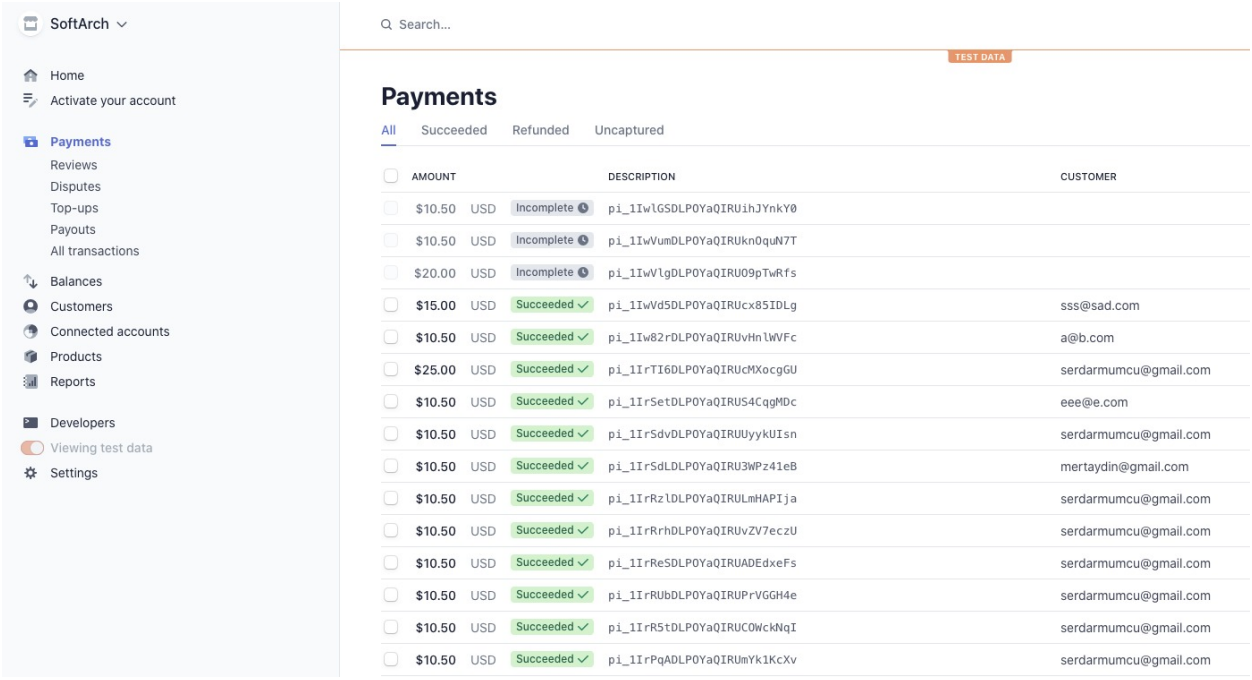
Solution	• Docker compose will be used for setting up the development environment *(Supportability Req. 3.4.2)*
Motivation/Justification	• Docker Compose will provide a standard development environment across the team. • Developers do not need to separately download and install each development tool and configure settings. • It will prevent "Works on my machine" excuse
Alternatives considered	Installing tools on the developer's host environment

Technical Memo-7	
Issue	Scaling the system if the usage load increases
Solution	• Docker Swarm will be used to handle scaling and encountering additional load *(Supportability Req. 3.2.3, 3.3.2 and 3.3.3)*
Motivation/Justification	• We are using a single machine for deployment in the development phase but we are considering the need for scaling from the beginning • Docker Swarm will provision extra machines if the load increases
Alternatives considered	Kubernetes

Technical Memo-8	
Issue	Payment integration
Solution	• We will use an external payment gateway
Motivation/Justification	• We integrated Stripe for the first release • It has its own interface and completely decoupled from our system • We can change the gateway or add other gateways in the future
Alternatives considered	Adyen, BKM

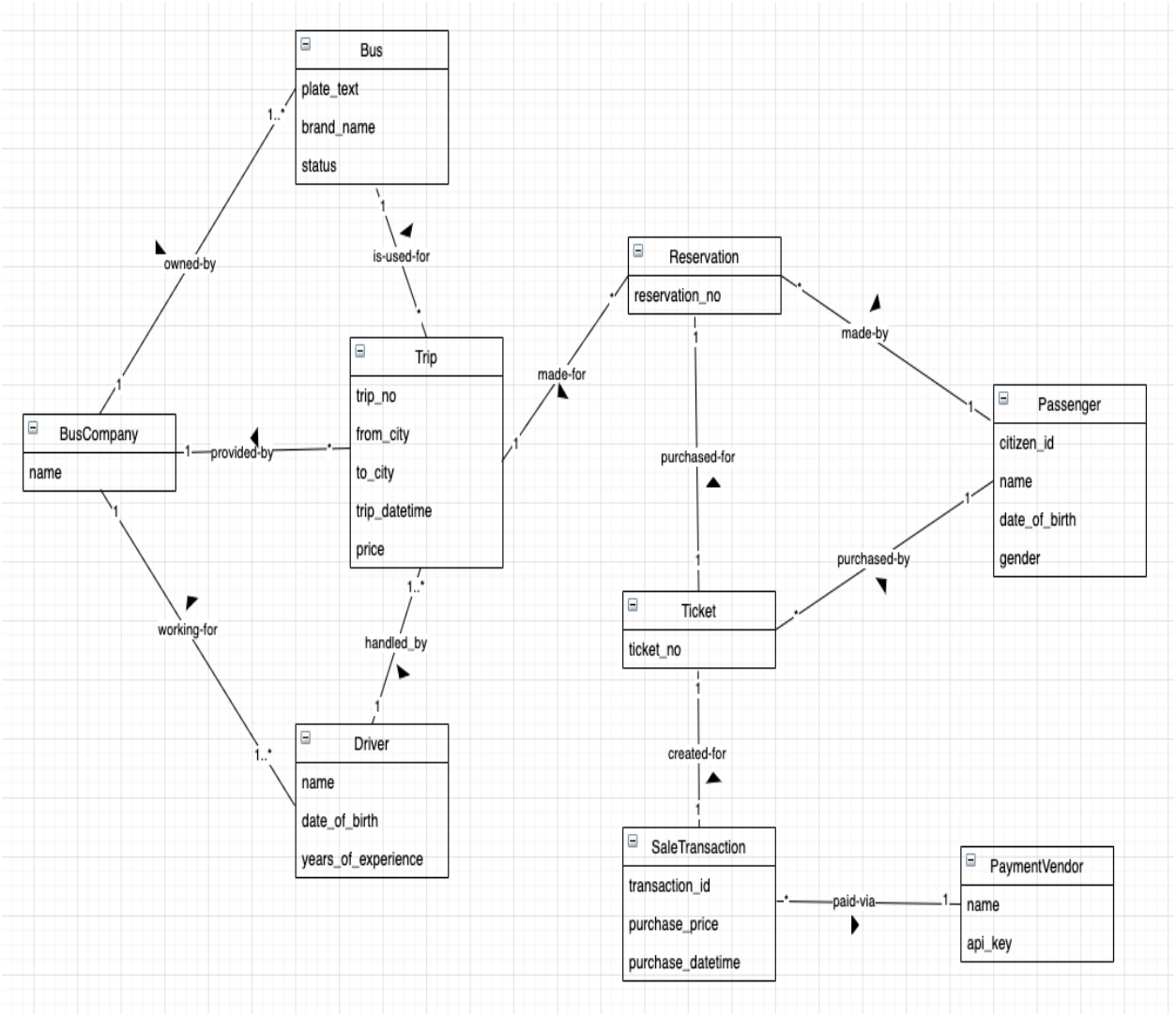
6. Architectural Mechanisms

1. Persistence mechanisms:
    a. SQLite relational database management system will be used as the persistence mechanism.
    b. SQLite is easy to work with as it works as an embedded database on the machine that runs the application rather than working as a client-server database.
2. Model-View-Template pattern:
    a. Django framework uses a variation of the Model-View-Controller (MVC) compound design pattern called the Model-View-Template (MVT) pattern.
    b. The Model in MVC is the same as the Model in MVT.
    c. The View in MVC is the same as the Template in MVT.
    d. The Controller in MVC is the same as the View in MVT.
3. Payment mechanism will be decoupled from the software. We will use Stripe as the payment gateway for the first release. Stripe is a SaaS tool for processing payments worldwide. We can add other payment gateways in future releases or we can support different payment infrastructures using the adapter design pattern. Stripe has an advanced management interface as shown in the below screenshot.



7. Key abstractions

Domain Model of the system is presented below. It includes the key abstractions from the problem domain.



8. Layers or architectural framework

The system will be constructed using a 3-tier layered architecture. Django uses MVT pattern instead of MVC pattern as explained in section 6 above. The layers of the system is summarized below:

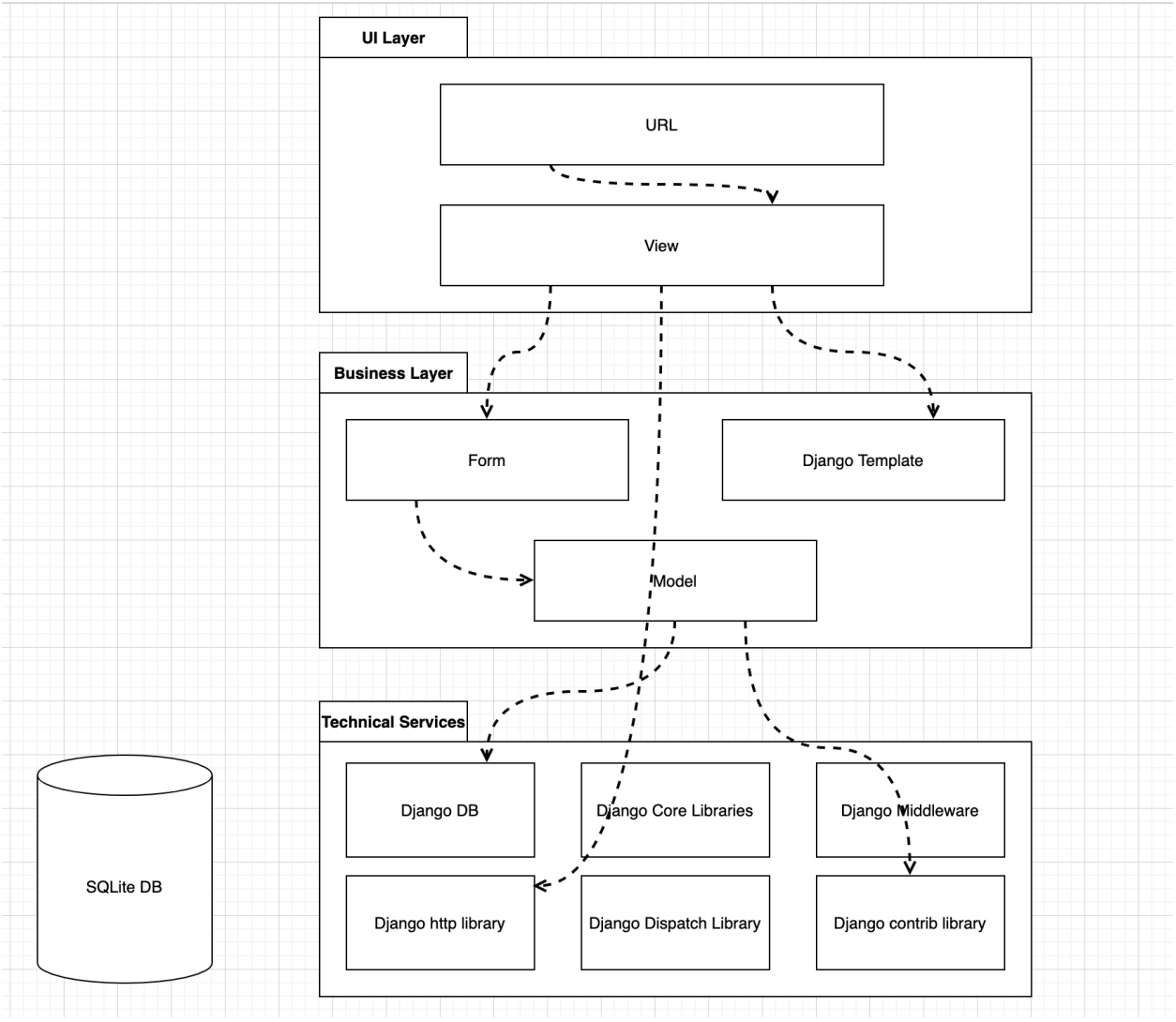
1. User interface: These are the views presented to the user. These views are HTML files mixed that are generated from Django templates.
2. Business Logic layer: This layer includes the Django views and Django models. These are equivalent to controllers.

3. Technical services (Core Django framework, modules, session management, signal handlers, http handlers, form mechanism, Django middleware, templates and shortcuts, generic views, web server creation, security features, other dependencies)

9. Architectural views

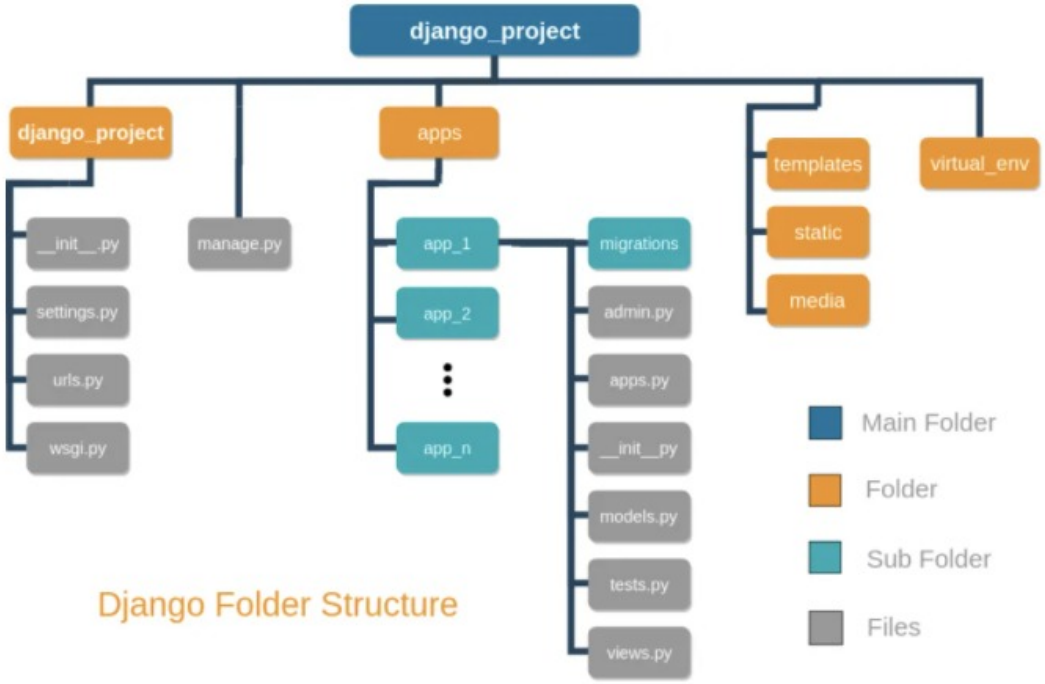
a. Logical View:

Logical view is presented below. It shows how MYES Bus Ticketing Application's different components are logically placed inside the 3-tier architecture.



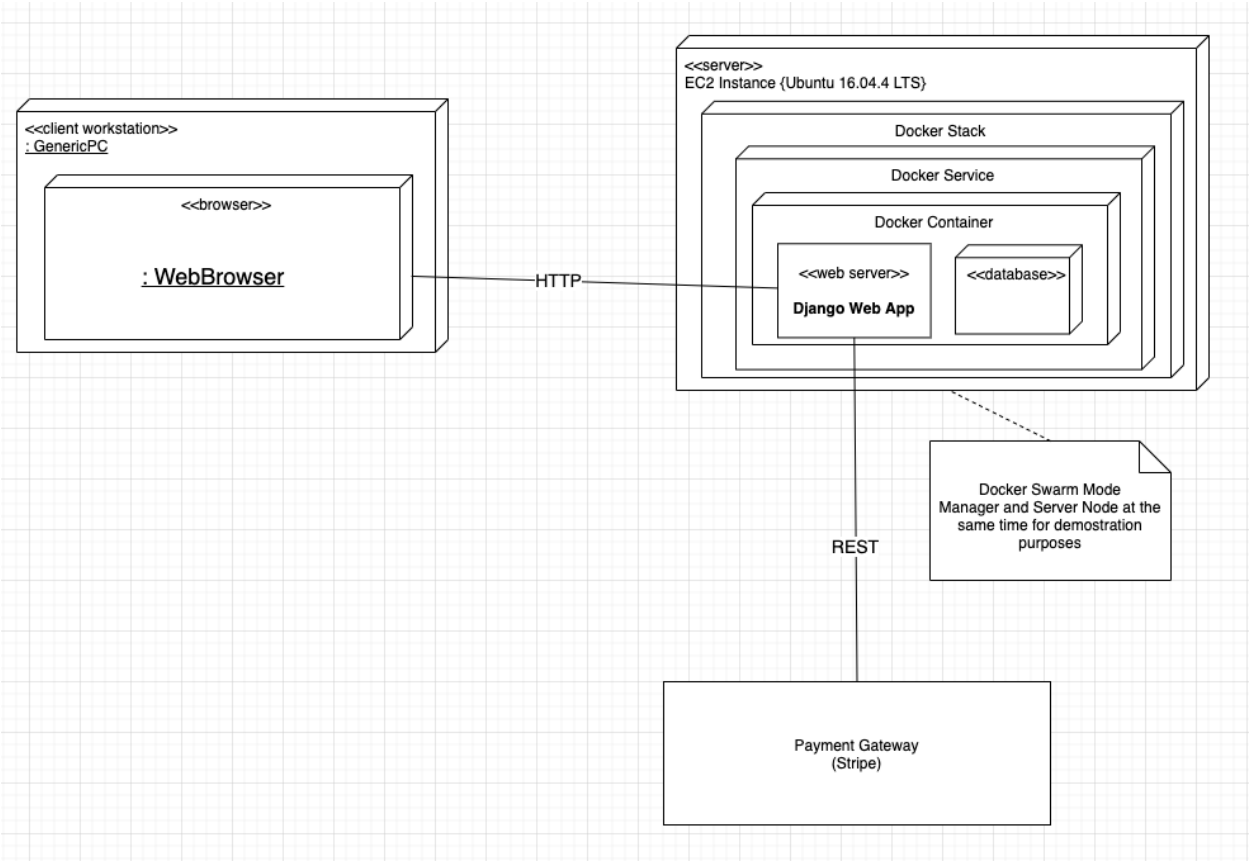
b. Implementation View:

Django web framework provides a standard folder structure for every web application. In our project source code, “django_project” is named as “myes” and “app” is named as “busticket”.

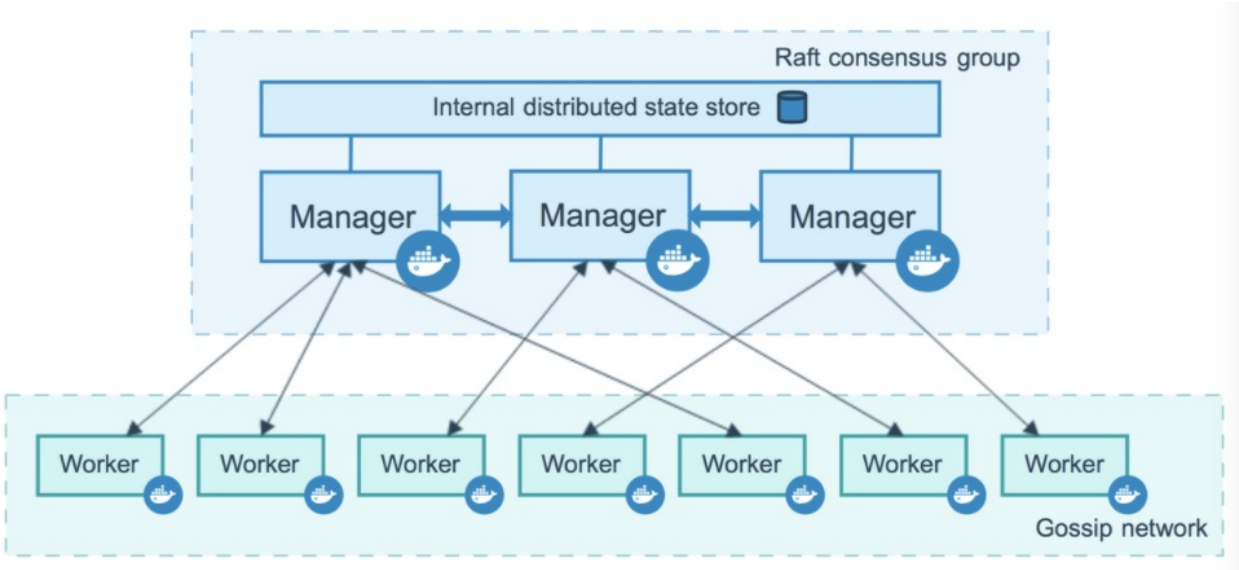


c. Deployment View:

Physical deployment of the application is shown in the diagram below. A use accesses the system using a web browser. Http requests made by the browser reaches the application that is running on a container which is running in an AWS ECS instance. Database is also placed in the same container.

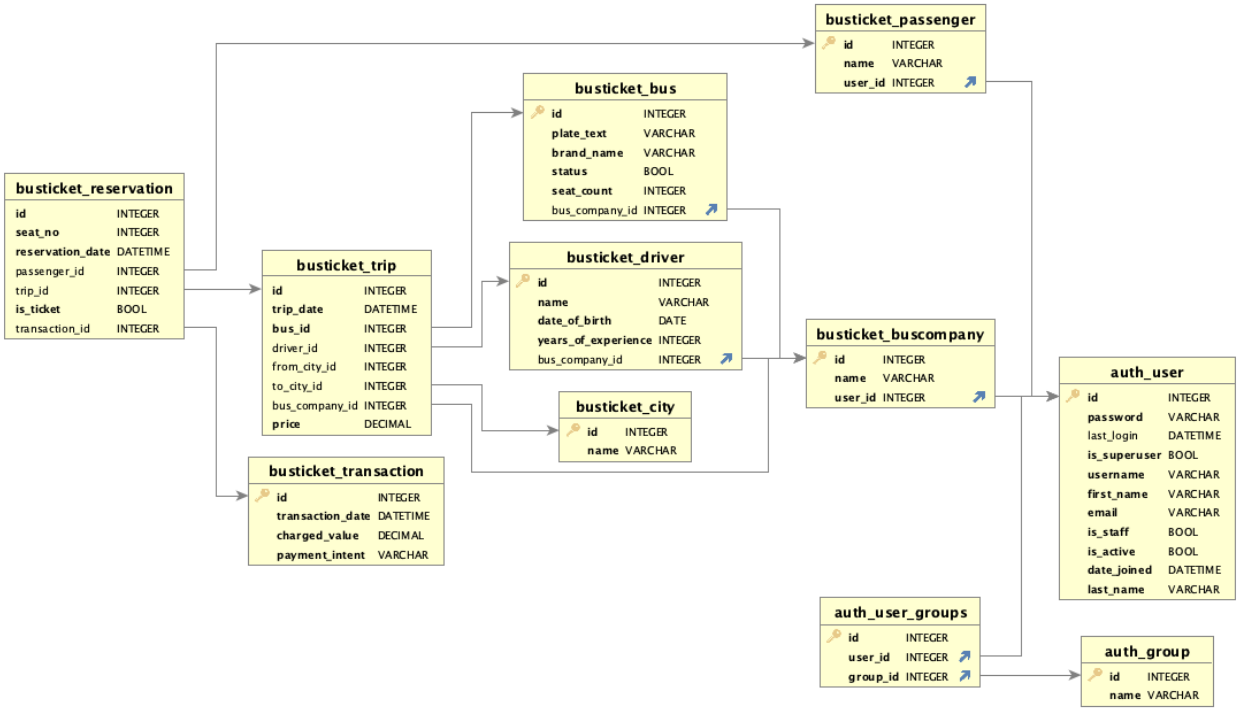


We are currently deploying a single virtual machine but we can scale up if needed. We are using Docker Swarm mode to orchestrate virtual machines and scale up or down when needed. In this type of Swarm mode, there are manager and worker nodes. This deployment style not only helps scaling but also good for fault tolerance. More information is provided in the Docker documentation.
(https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/)



d. Data View:

The Entity-relationship diagram shows the data view of the application below.



e. Use Case View: