Use Case Study Report

Group No.: Group 04

Student Names: Ronit Mankad and Shashank Nukala

Executive Summary:

The objective of this study was to design and implement a production ready relational database that can be implemented in a real world Appstore environment. With the exploding number of mobile users, fault tolerant and reliable application distribution architectures become essential. The modern Appstore is its latest evolution.

The database used in this study was modelled by analyzing existing Appstore architecture like Apple's Appstore and Google's Playstore. All the requirements were carefully collected to develop the ER and UML diagrams. These conceptual models successfully modeled the use case requirements. They were then mapped to a relational model where the requirements and constraints were satisfied by using clever combinations of primary keys, foreign keys and relations. Moreover, the relational model was normalized to reduce redundancy and increase reliability. This database was then fully implemented in MySQL. The database was then connected to a Flask web application written in Python which accessed the database using the Flask-MySQL library. Dashboards were incorporated in this web app which can be used by a DBA to perform CRUD operations.

This implementation turned out to be highly scalable and fault tolerant. The biggest improvement over MySQL's default constraint measures was the constraint checking and form validation features of the Flask-Forms library. A web form interface is more user friendly and makes it easy to perform CRUD operations. In the future, the functionality of this DBMS can be enhanced by including more granular querying options and incorporating more complex relationships and entities.

I. Business Problem definition and requirement

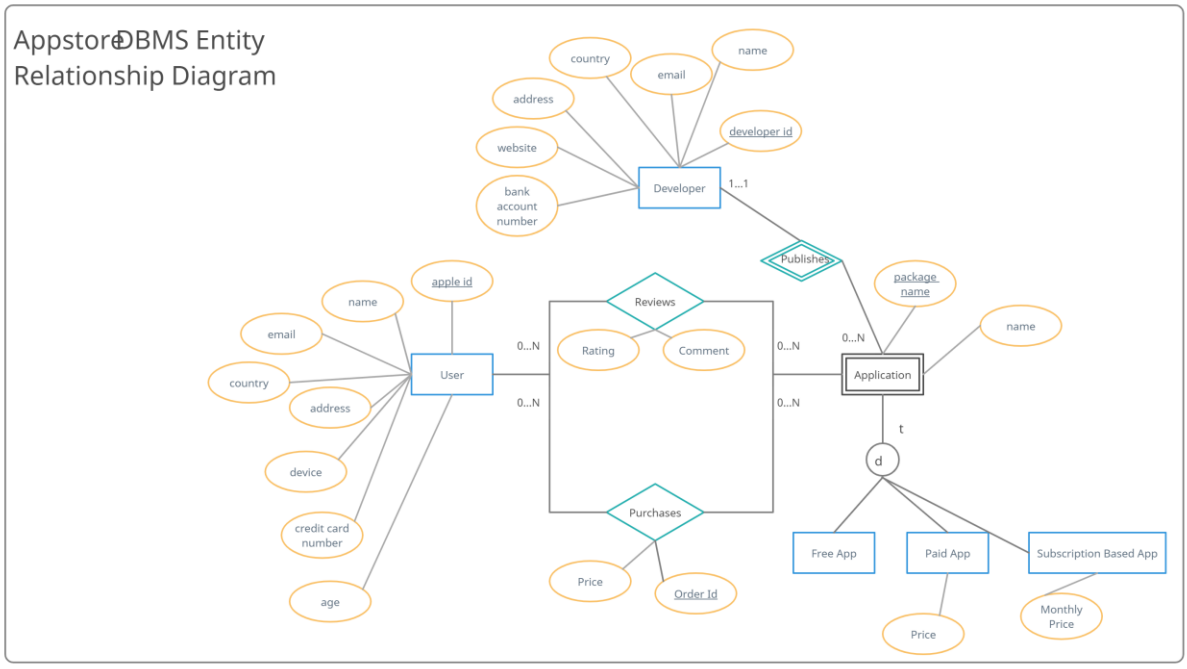
Apple is in the need of a new database system for it's Appstore for the new version of iOS. In the Appstore, developers publish their pre-approved apps. Alongside the existing purchase model, the new Appstore should also support the new subscription-based model introduced by Apple where users can subscribe for applications. An application can be of three types – a free app, a paid app or an app with a subscription model. The Appstore should allow the user to download any of these three types of applications. The new Appstore should also limit users to be able to download apps available in their country. Apple earns a 30% cut of the app price and the rest goes to the developer. Users should be able to review an application by giving it a rating between 1 to 5 stars and by making a comment.

The database requirements are:

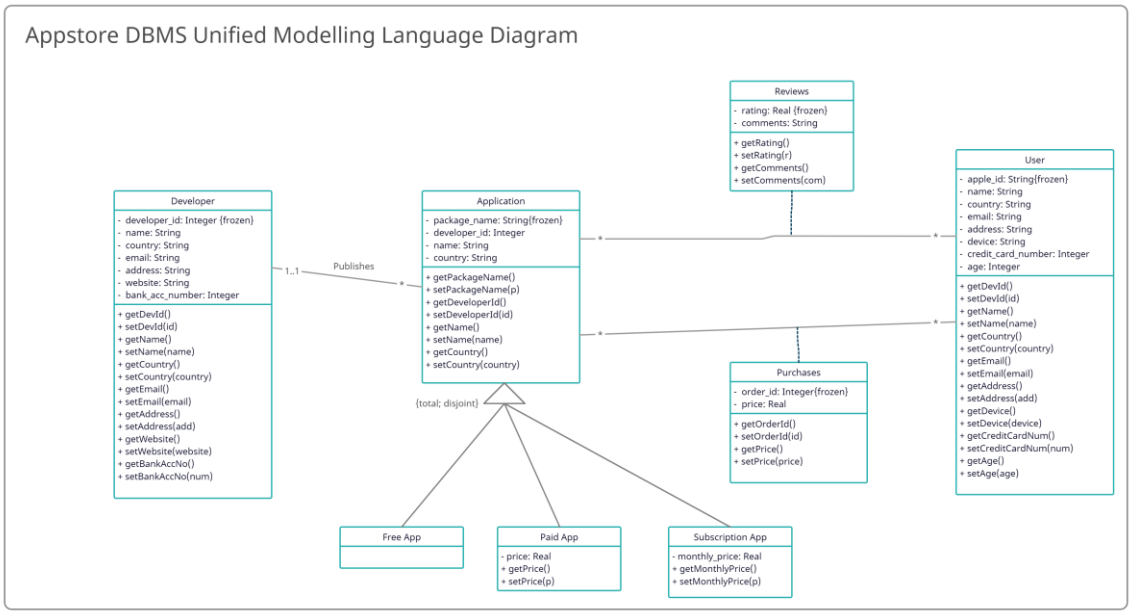
1. A developer can publish multiple apps. Each developer has a unique developer id.
2. Each application has its unique package name which is used to identify the application. Multiple applications can have the same display name, but the package name is always unique.
3. Users are identified by their unique apple id.
4. A user can rate an app between 0 to 5 stars. User can also provide comments with their rating.
5. A user pays with their credit/debit card linked with their apple id.

II. Conceptual Data Modeling

Enhanced Entity Relationship (EER) Diagram



Unified Modelling Language (UML) Diagram



III. Mapping Conceptual Model to Relational Model

Relational Model

- Developer (developer_id, name, email, country, website, address, bank_acc_number)
- User (apple_id, name, email, country, address, device, credit_card_num, age)
- Application (package_name, *developer_id*, name, app_type, price)
  app_type constraint will be enforced in the programming language
- Purchases (order_id, *apple_id*, *app*, price, purchase_date)
  Foreign Key *app* refers to *package_name* in Application; NOT NULL
  Foreign Key *apple_id* refers to *apple_id* in User; NOT NULL
- Reviews(*user*, *app*, rating, comments)
  Foreign Key *app* refers to *package_name* in Application; NOT NULL
  Foreign Key *user* refers to *apple_id* in User; NOT NULL
  User and *app* together make up the foreign key of this relation.

IV. Implementation of Relational Model via MySQL and Flask

MySQL Implementation

The database was implemented in MySQL using MySQL Workbench. Here are the DDL statements:

```
create database AppStore;
use Appstore;

-- Create Developers
create table
developers(developer_id bigint not null,
           name varchar(20) not null,
           email varchar(30) unique not null,
           country varchar(20) not null,
           address varchar(100) not null,
           website varchar(30) not null,
           bank_acc_number bigint not null,
           primary key(developer_id)
           );

-- Create Users
create table
users(apple_id varchar(30) not null,
           name varchar(20) not null,
           email varchar(30) not null,
           country varchar(20) not null,
           address varchar(100) not null,
           device varchar(30) not null,
           credit_card_num bigint not null,
           age integer not null,
           primary key(apple_id)) ;
```

```
-- Create Applications
create table applications(package_name varchar(30) not null,
            developer_id bigint not null,
            name varchar(30) not null,
            app_type varchar(30)  not null,
            price float not null,
            primary key(package_name),
            foreign key(developer_id) references developers(developer_id) on delete cascade);

-- User Purchases an app
create table purchases(order_id bigint unique not null,
                apple_id varchar(30) not null,
                app varchar(30) not null,
                price real not null,
                purchase_date date not null,
                primary key(apple_id, app),
                foreign key(apple_id) references users(apple_id) on delete cascade,
                foreign key(app) references applications(package_name) on delete cascade
                );

-- User Reviews and app
create table reviews(user varchar(30) not null,
                     app varchar(30) not null,
                     rating real not null,
                     comment varchar(30),
                     primary key(user, app),
                     foreign key(user) references users(apple_id) on delete cascade,
                     foreign key(app) references applications(package_name) on delete cascade
                    );
```

Python/Flask Implementation

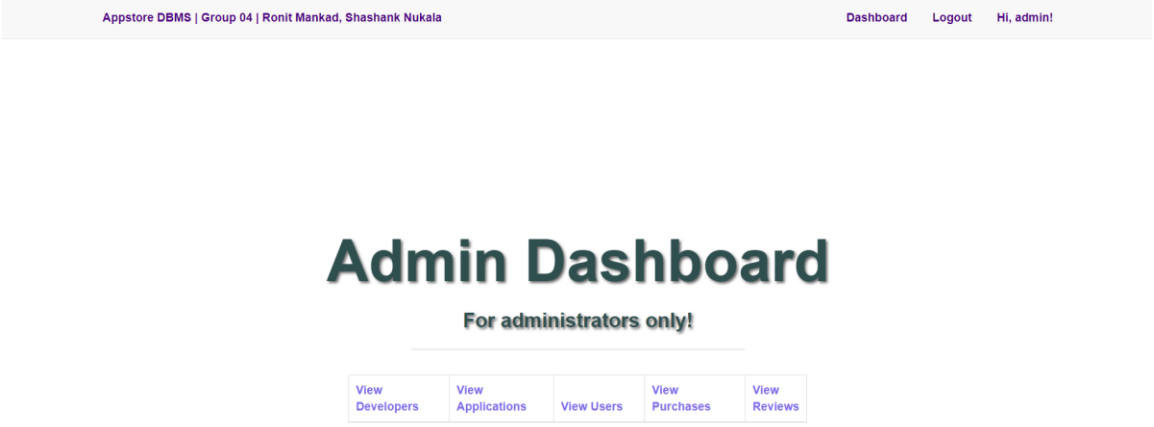
The MySQL database was then connected with a Flask Web app server. Here is a short snapshot of a Flask view showing all developers in the database:

```
# developer Views
@home.route('/developers', methods=['GET', 'POST'])
@login_required
def list_developers():
    """
    List all developers
    """
    db = MySQLdb.connect("localhost", 'AppstoreDB', 'appstore12345', 'Appstore')

    cur = db.cursor()

    query_string = "SELECT * from developers;"
    cur.execute(query_string)

    developers = cur.fetchall()

    db.close()


    return render_template('home/developers/developers.html',
                           developers=developers, title="Developers")
```

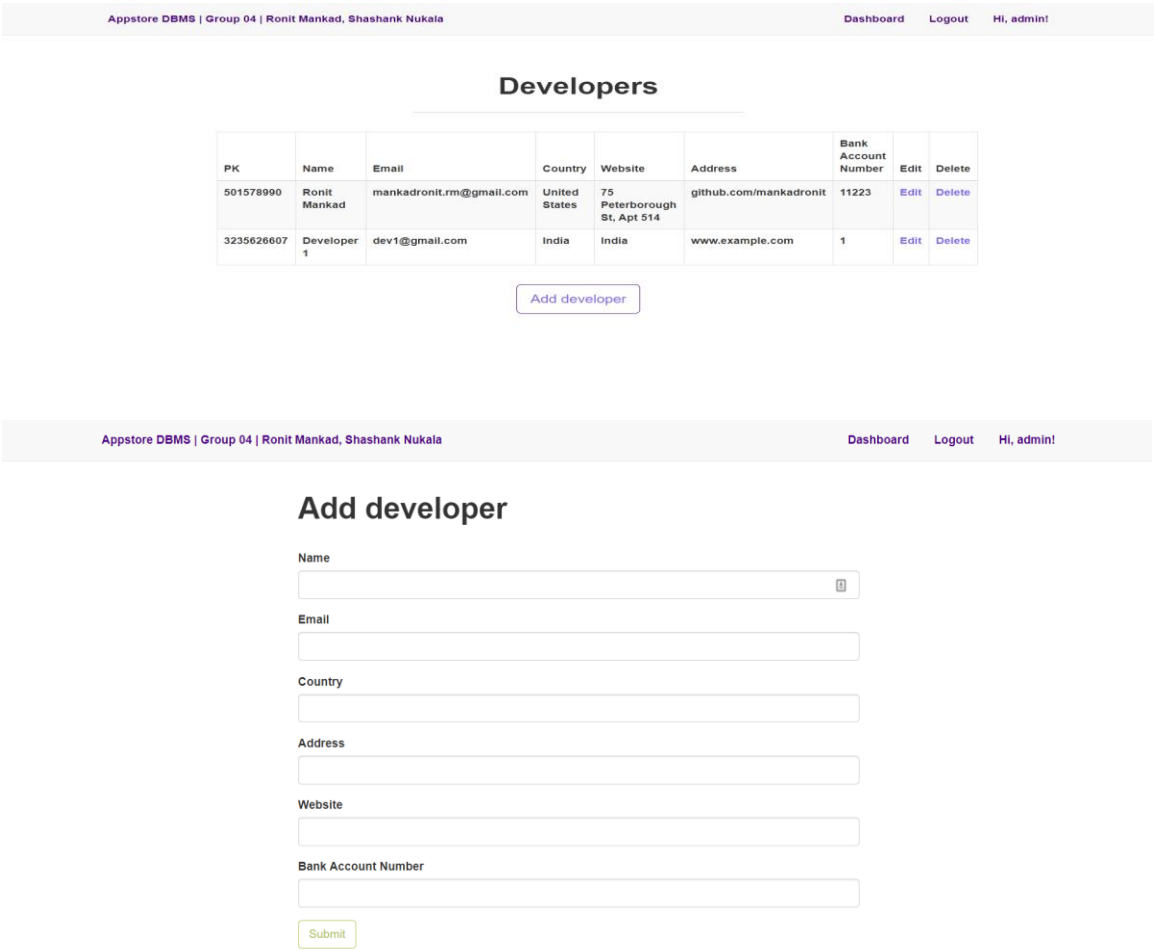
As seen from the code, the code first established a connection with the database using the MySQLdb.connect() method passing the host, username, password and database name as arguments. It then selects all developers from the developers table and passes them to the developers.html template which displays it.

Different Flask Dashboards

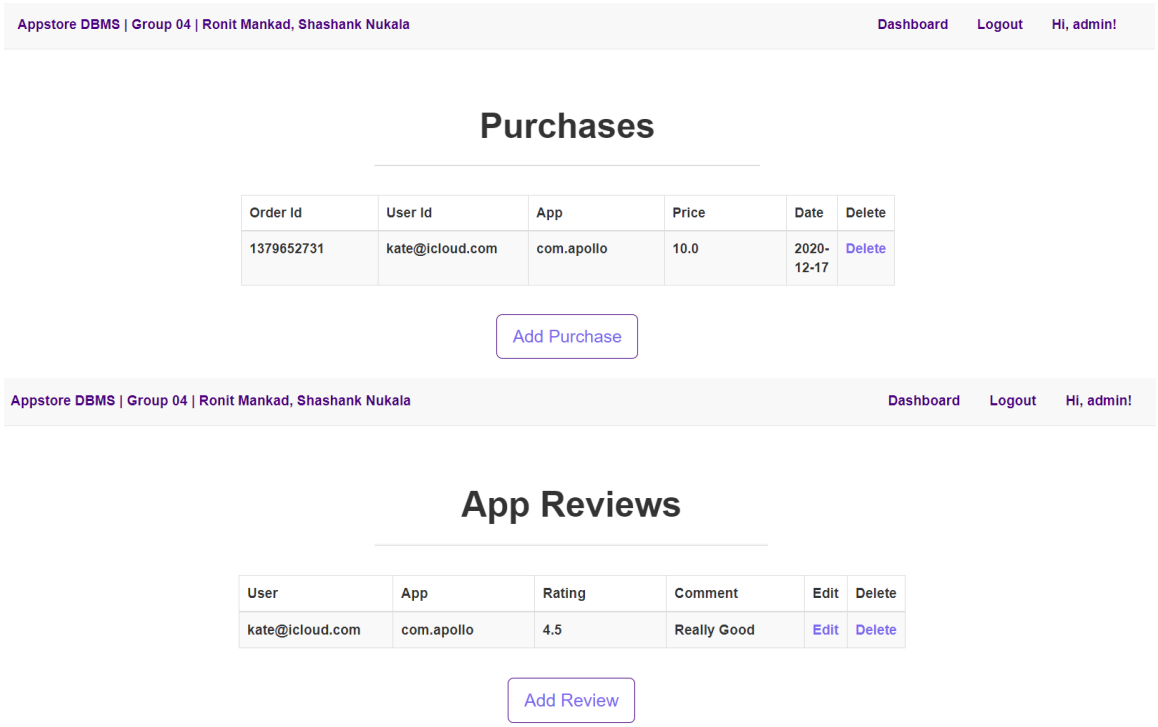
Main Dashboard



Developers Dashboard and Add Developer Form

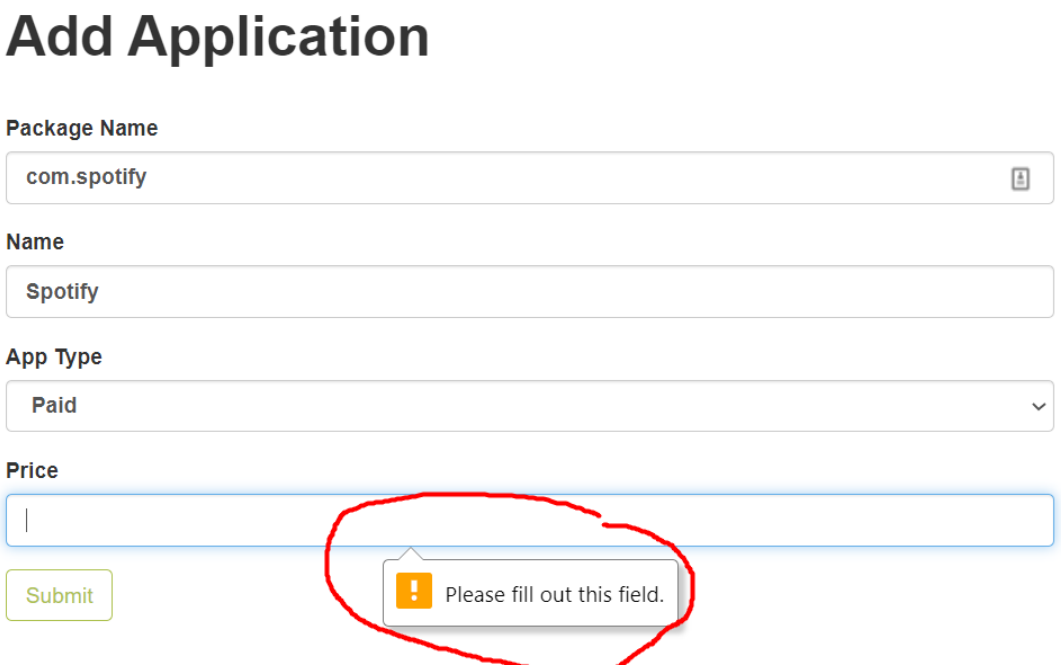


Viewer Purchases and Reviews



Data Validation and Constraint Check

Price of a paid app cannot be empty and hence the form throws an error. Validations and constraint checking like this makes this app highly fault tolerant.



V. Summary

This implementation turned out to be highly scalable and fault tolerant. The biggest improvement over MySQL's default constraint measures was the constraint checking and form validation features of the Flask-Forms library. A web form interface is more user friendly and makes it easy to perform CRUD operations. In the future, the functionality of this DBMS can be enhanced by including more granular querying options and incorporating more complex relationships and entities.