DCU EEN1037 -Assignment 3

Server-Side Programming

Name: Mohammed Al Shuaili

Date: 15/04/2025

Table content

[Introduction 3](#_Toc195655637)

[Implementation 3](#_Toc195655638)

[Django Models / SQL Database Tables 3](#_Toc195655639)

[A. 4 or More SQL Database Tables 3](#_Toc195655640)

[B. ForeignKey Relationship Implementation: 3](#_Toc195655641)

[C. Migrations and SQL Table Creation 4](#_Toc195655642)

[Forms for typical CRUD (Create, Read, Update, Delete) operations 5](#_Toc195655643)

[a. 4+ POST Request Forms: 5](#_Toc195655644)

[b. Visibility of Changes: 5](#_Toc195655645)

[c. Server-Side Validation 5](#_Toc195655646)

[d. Client-Side JavaScript Validation 6](#_Toc195655647)

[Client-side AJAX Functionality 6](#_Toc195655648)

[a. JSON-Based AJAX GET Request 6](#_Toc195655649)

[b. JSON-Based AJAX POST Requests 7](#_Toc195655650)

[c. Integration with Django Views 8](#_Toc195655651)

[d. Error Handling and User Feedback 8](#_Toc195655652)

[User registration & login functionality 8](#_Toc195655653)

[a. User Registration 8](#_Toc195655654)

[b. User Login 9](#_Toc195655655)

[c. User Logout 9](#_Toc195655656)

[d. Staff User Handling 10](#_Toc195655657)

[Docker container, SQL Database connectivity & migrations 10](#_Toc195655658)

[a. Docker Configuration 10](#_Toc195655659)

[b. Database Connectivity 11](#_Toc195655660)

[Conclusion 12](#_Toc195655661)

# Introduction

AutoGadget Hub is an e-commerce platform designed to provide automotive enthusiasts and tech-savvy consumers with cutting-edge gadgets, tools, and accessories. Targeting car owners, DIY mechanics, and tech lovers, the platform offers a seamless shopping experience with features such as dynamic product catalogs, secure checkout, and personalized order tracking. This assignment demonstrates the implementation of a full-stack web application using Django, focusing on functionalities such as database design, user authentication, CRUD operations, and AJAX-driven interactions, all containerized for scalable deployment.

# Implementation

# Django Models / SQL Database Tables

## A. 4 or More SQL Database Tables

The code defines six Django models, each representing a SQL database table:

1. **UserProfile**: Extends Django’s built-in User model to store additional profile data (address, phone number).
2. **ProductCategory**: Represents product categories (e.g., "Electronics", "Clothing").
3. **Product**: Stores product details like name, price, stock, and links to a category via a foreign key.
4. **CartItem**: Tracks items added to a user’s shopping cart, including quantity and product references.
5. **Order**: Manages order metadata (delivery address, total price, status).
6. **OrderItem**: Represents individual items within an order, including quantity and price.

These models are mapped to SQL tables by Django’s ORM. For example, the Product model includes fields like price and stock, which translate to DECIMAL and INTEGER SQL columns, respectively.

## B. ForeignKey Relationship Implementation:

The code includes multiple foreign key relationships. An example is the Product model’s category field which establishes a many-to-one relationship where a Product belongs to a ProductCategory. If the category is deleted, the product’s category is set to NULL to preserve data integrity. All the relationships are as outlined in the scheme below:

A diagram of a product

AI-generated content may be incorrect.

## C. Migrations and SQL Table Creation

The models are accompanied by Django migrations (stored in the migrations/\*.py files), which automate SQL table creation. For instance:

* Running python manage.py makemigrations generates migration files based on the model definitions.
* Executing python manage.py migrate applies these migrations, creating SQL tables like models\_product, models\_order, etc., with appropriate columns and constraints.

For example, the Product model’s price field becomes a DECIMAL(10, 2) column in SQL, and foreign keys (e.g., category\_id in Product) are implemented as INTEGER columns referencing related tables as shown below. Migrations ensure the database schema evolves alongside the code, eliminating manual SQL scripting.

A screen shot of a computer

AI-generated content may be incorrect.

# Forms for typical CRUD (Create, Read, Update, Delete) operations

## a. 4+ POST Request Forms:

The code implements six POST forms for CRUD operations, covering all required actions:

1. Create:

* ProductForm: Adds new products (fields: category, name, price, etc.).
* ProductCategoryForm: Creates product categories.
* UserRegistrationForm: Registers new users with profile data.

1. Update:

* ProductEditForm: Modifies existing products (e.g., price, stock).

1. Delete:

* delete\_product (via view): Removes a product using POST.
* category\_delete (via view): Deletes a category if no products are linked.

**Example Workflow for Create (ProductForm):**

* A staff user submits the ProductForm (HTML form with method="POST").
* Django validates fields server-side (e.g., clean\_price() ensures positive values).
* On success, the product appears in the manage\_products page.

## b. Visibility of Changes:

After CRUD operations, changes are reflected in relevant pages:

* Create/Update: New or edited products appear in manage\_products (staff view) and the public shop page.
* Delete: Removed products/categories vanish from listings.
* User Registration: New users can log in immediately, and their profiles are accessible via UserProfile.

## c. Server-Side Validation

All forms enforce validation logic in forms.py. For example:

* In ProductForm:

def clean\_price(self):

if price <= 0:

raise ValidationError("Price must be > 0.") # Server-side check

* In UserRegistrationForm:
  1. Checks for duplicate usernames/emails in clean\_username() and clean\_email().
  2. Validates password matching in clean().
* In checkout\_form:
  1. Uses Luhn algorithm for credit card validation and expiry date checks.

## d. Client-Side JavaScript Validation

Two forms include client-side validation:

1. **ProductForm** (product\_validation.js):
   * Validates name length (≥3 chars), price (>0), and stock (≥0) before submission.
   * Displays inline errors (e.g., "Price must be positive").
2. **UserRegistrationForm** (signup.js):
   * Enforces password complexity (8+ chars, special characters).
   * Validates email format and phone number structure.

The Figure below shows server and client side validation in the same form.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

# Client-side AJAX Functionality

## a. JSON-Based AJAX GET Request

Implementation:

The order\_history.html template includes an AJAX GET request to fetch order data dynamically:

// order\_history.html

fetch(`{% url "order\_history\_api" %}?period=${encodeURIComponent(period)}`)

.then(response => response.json())

.then(data => renderOrders(data.orders))

The purpose of using it in this place is to retrieve order history for a selected time period (day, month, or year). The `order\_history\_api` endpoint in `views.py` returns JSON data containing the relevant order details. The `renderOrders()` function dynamically updates the DOM to display the fetched orders without requiring a page reload. In case the request fails, an error message is displayed while ensuring the rest of the page remains functional.

## b. JSON-Based AJAX POST Requests

**implementation:**

**1. Add to Cart (shop.html):**

// shop.html

await fetch(form.action, {

method: 'POST',

headers: {

'X-Requested-With': 'XMLHttpRequest',

'X-CSRFToken': csrfToken

},

body: formData

});

This action adds a product to the cart and dynamically updates the cart count in the header using data.cart\_count from the JSON response. If the stock is insufficient or the request fails, an alert is shown to inform the user.

**2. Update Cart Quantity (cart.html):**

// cart.html

await fetch(url, {

method: 'POST',

headers: {

'X-Requested-With': 'XMLHttpRequest',

'X-CSRFToken': csrfToken

},

body: formData

});

This action updates the item quantity in the cart or removes items by setting the quantity to 0. It dynamically updates the item totals (`data.item\_total`), cart total (`data.cart\_total`), and cart count, and removes the item's row from the DOM if the quantity is 0. In case of a failure, the input field is reset, and an alert is displayed to inform the user.

## c. Integration with Django Views

* **GET Example (Order History):**

The order\_history\_api view filters orders based on the period and user, and then returns JSON:

# views.py

def order\_history\_api(request):

………….

return JsonResponse({'orders': orders\_data})

* **POST Example (Cart Updates):**

The update\_cart view handles both AJAX and regular POST requests, returning JSON for AJAX:

# views.py

def update\_cart(request, product\_id):

……..

return JsonResponse(response\_data) # For AJAX requests

## d. Error Handling and User Feedback

* GET Errors:

fetch(url).catch(error => {

errorMessage.textContent = 'Error retrieving orders: ' + error.message;

errorMessage.style.display = 'block';

});

* POST Errors:

catch (error) {

alert('Error: ' + error.message);

quantityInput.value = quantityInput.defaultValue; // Reset input

}

# User registration & login functionality

## a. User Registration

Implementation:

* UserRegistrationForm (forms.py):
* A custom form collects username, email, password, address, and phone number. Key features:

class UserRegistrationForm(forms.Form):

# Fields for username, email, password, address, phone, etc.

def clean\_username(self):

if User.objects.filter(username=username).exists():

raise ValidationError("Username taken") # Server-side check

* Client-Side Validation: signup.js enforces username format, password complexity, and phone number structure.
* UserProfile Creation: On valid submission, signup\_view creates both a User and a linked UserProfile:

# views.py

User.objects.create\_user(...)

UserProfile.objects.create(user=user, address\_line1=..., phone\_number=...)

**Templates:**

* signup.html renders the form with error messages. Users are redirected to login after registration.

## b. User Login

**Implementation:**

* Login View (signin\_view in views.py):

Uses Django’s built-in authenticate() and login():

user = authenticate(request, username=username, password=password)

if user is not None:

login(request, user)

return redirect('home')

* Template: signin.html provides a simple login form. Failed attempts show error messages.

## c. User Logout

**Implementation:**

* **Logout View (**signout\_view**):**

logout(request)

messages.success(request, 'Logged out successfully')

return redirect('home')

* Template: The "Sign Out" link is visible in the header for authenticated users.

## d. Staff User Handling

**Implementation:**

* **Staff-Only Views:**  
  Views like manage\_products and add\_product are protected with @staff\_member\_required:

@staff\_member\_required

def manage\_products(request):

# Staff-only logic

* Template Logic:

{% if user.is\_staff %}

<a href="{% url 'manage\_products' %}">Manage Products</a>

{% endif %}

* Staff Creation: Regular users are promoted to staff via Django Admin by setting is\_staff=True.

# Docker container, SQL Database connectivity & migrations

## a. Docker Configuration

**Implementation:**

in order to configure the docker as per the requirements, 3 main files are added

1. Dockerfile:

This Docker setup uses Python 3.13 as the base image, installs dependencies from `requirements.txt` (ensuring `mysqlclient` is included for MySQL support), configures essential environment variables such as `DATABASE\_URL` and superuser credentials, and runs the `docker\_entrypoint.sh` script on startup.

1. **Entrypoint Script (**docker\_entrypoint.sh**):**

On startup, the container automatically runs database migrations, creates a Django admin user if the relevant credentials are provided, and starts the Django development server.

#!/bin/sh

python manage.py migrate --noinput # Auto-run migrations

python manage.py createsuperuser # Create admin if credentials are provided

python manage.py runserver 0.0.0.0:8000

1. .dockerignore**:**

Excludes unnecessary files (e.g., IDE configs, virtual environments, local databases).

## b. Database Connectivity

**Implementation:**

1. dj-database-url Integration:

In settings.py, the database is configured dynamically using the DATABASE\_URL environment variable:

DATABASES = {

'default': dj\_database\_url.config(

default='sqlite:///...', # Fallback to SQLite

conn\_max\_age=600

)

}

This supports MySQL, PostgreSQL, or SQLite based on the provided DATABASE\_URL.

1. **MySQL connection**

When running the container using the following code (as provided in the assignment):

docker run -ti \

-e DATABASE\_URL="mysql://myappdbuser:myappdbpass@host.docker.internal:3306/myappdb" \

-v myapp-storage:/app/storage \

-p 8000:8000 myapp

It will connect to a MySQL server on the host machine using mysqlclient (in requirements). In addition, A readme and readme\_MYSQL files are provided for user instructions on how to run it in a docker container, the following outputs show all the tables created in MYSQL shell:

A screenshot of a computer program

AI-generated content may be incorrect.

# Conclusion

This project successfully delivers an e-commerce platform leveraging Django’s ORM, form handling, and authentication system. Key features include relational database models, client/server-side validation, AJAX for dynamic updates, and Docker for environment consistency. The implementation highlights adherence to best practices in security, usability, and scalability. Through this assignment, I gained expertise in integrating Django with modern tools like Docker and MySQL, while refining my ability to design user-centric web applications.