DCU EEN1037 -Assignment 3

Server-Side Programming

Name: Mohammed Al Shuaili

Date: 15/04/2025

Table content

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

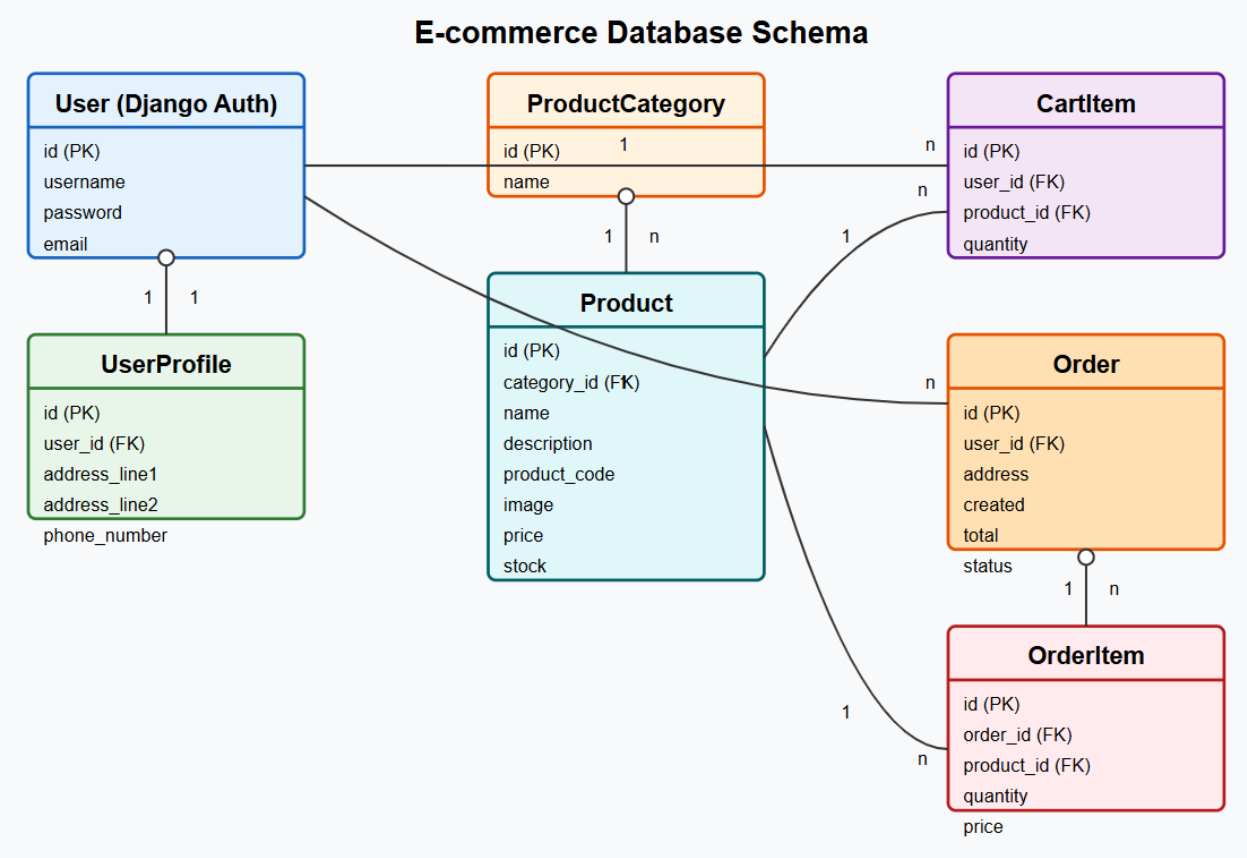
The code defines six Django models, each representing a SQL database table:

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:



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.

```
+--------------+----------------+------+-----+---------+----------------+
| Field        | Type           | Null | Key | Default | Extra          |
+--------------+----------------+------+-----+---------+----------------+
| id           | bigint         | NO   | PRI | NULL    | auto_increment |
| name         | varchar(200)   | NO   |     | NULL    |                |
| description  | longtext       | NO   |     | NULL    |                |
| product_code | varchar(50)    | NO   | UNI | NULL    |                |
| image        | varchar(100)   | NO   |     | NULL    |                |
| price        | decimal(10,2)  | NO   |     | NULL    |                |
| stock        | int unsigned   | NO   |     | NULL    |                |
| created_at   | datetime(6)    | NO   |     | NULL    |                |
| category_id  | bigint         | YES  | MUL | NULL    |                |
+--------------+----------------+------+-----+---------+----------------+
```

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.

2. Update:

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

3. 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
```

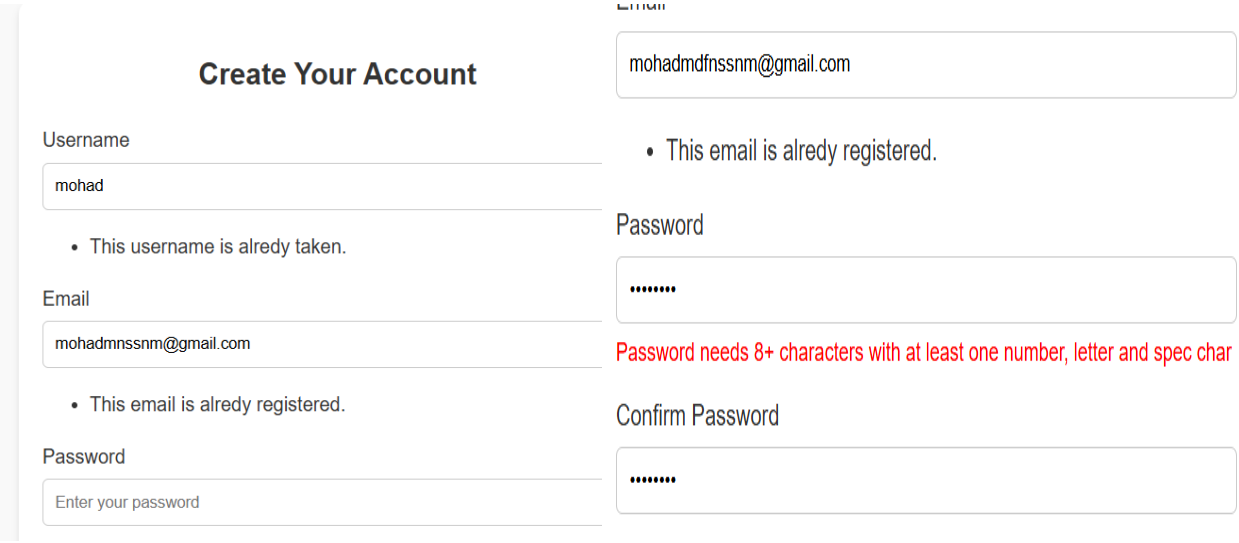
- In UserRegistrationForm:
    1. Checks for duplicate usernames/emails in clean_username() and clean_email().
    2. Validates password matching in clean().
- In checkout_form:
    1. Uses Luhn algorithm for credit card validation and expiry date checks.

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.



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.

2. **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
```

3. .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.

2. **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:

```
mysql> SHOW tables;
+----------------------------+
| Tables_in_myappdb          |
+----------------------------+
| auth_group                 |
| auth_group_permissions     |
| auth_permission            |
| auth_user                  |
| auth_user_groups           |
| auth_user_user_permissions |
| django_admin_log           |
| django_content_type        |
| django_migrations          |
| django_session             |
| myapp_cartitem             |
| myapp_order                |
| myapp_orderitem            |
| myapp_product              |
| myapp_productcategory      |
| myapp_userprofile          |
+----------------------------+
16 rows in set (0.01 sec)
```

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.