Backend

For our backend development, we leverage the robust Python framework known as Django. Renowned for its elegance and efficiency, Django empowers developers to craft web applications swiftly and seamlessly. Embracing the ethos of 'batteries-included,' Django furnishes a rich toolkit replete with pre-built functionalities, sparing developers the need for tedious boilerplate code. Its hallmark lies in its commitment to a clean and pragmatic design, fostering a development environment that is both professional and user-friendly. With Django at our disposal, we navigate the complexities of backend development with precision and finesse, ensuring our applications are not only sophisticated but also impeccably crafted

Here what Django Python encompasses:

Django, a high-level Python web framework, embodies the synergy of Python's simplicity and versatility. It follows the MVC architectural pattern, promoting clean and pragmatic design principles. With Django, developers access a rich suite of built-in tools and libraries, including an ORM for database interactions, a URL dispatcher for routing requests, and an authentication system. The framework's vibrant community fosters collaboration and innovation, supported by a robust ecosystem of third-party packages. In essence, Django empowers developers to rapidly build web applications with precision and efficiency, making it an ideal choice for projects of any scale

It simplifies the development of web applications by providing a comprehensive set of tools and conventions that include:

1. Incoming requests from clients are managed by Django's middleware, which passes them to the URL dispatcher for routing.

This describes the process in Django where incoming requests from clients, such as web browsers, are first intercepted by Django's middleware. The middleware then forwards these requests to the URL dispatcher, which maps them to the appropriate view functions or classes for further processing. This is a crucial step in the request-response cycle of Django applications, allowing for efficient routing and handling of incoming requests.

2.URL ROUTING:

URL routing in Django maps URLs to corresponding view functions or classes, facilitating organized request handling and allowing for clear navigation within the web application. Each URL pattern defined in urlpatterns directs incoming requests to specific views, providing a structured approach to request handling and page rendering.

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
from store import views
```

This part of the code imports necessary modules and resources for URL routing, including the path function from django.urls, settings from django.conf, and the views module from your store app.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('mens', views.mens, name='mens'),
    path('women', views.women, name='women'),
    path('about', views.about, name='about'),
    path('registeration', views.registration, name='registration'),
    path('admin', admin.site.urls),
    path('accounts', include('django.contrib.auth.urls')),

] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

3.view processing

view processing in Django involves handling incoming requests, interacting with models to fetch or manipulate data, and rendering templates to generate dynamic HTML responses. Each view function corresponds to a specific URL pattern and is responsible for processing a particular type of request, interacting with models or forms as needed, and rendering the appropriate template to provide a response to the client

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from .forms import UserSignupForm, UserProfileForm
```

This part of the code imports necessary modules and resources for view processing, including functions from django.shortcuts, the login_required decorator from django.contrib.auth.decorators, and forms from the local forms.py file.

```
@login_required
def product_list(request):
    products = Product.objects.filter(available=True)
    return render(request, 'shop/product_list.html', {'products': products})
```

In this view function (product_list), the login_required decorator ensures that only authenticated users can access the view. The function interacts with the Product model to fetch a list of available products and passes them to the shop/product_list.html template for rendering

```
def index(request):
    return render(request, 'index.html')
```

The index view function simply renders the index.html template, which serves as the homepage of the web application..

```
def women(request):
    return render(request, 'women.html')
```

Similarly, the women view function renders the women.html template, which presumably displays content related to women's products or categories.

```
def mens(request):
    return render(request, 'men.html')
```

The mens view function renders the men.html template, which presumably displays content related to men's products or categories.

```
def about(request):
    return render(request, 'about.html')
```

The about view function renders the about.html template, which provides information about the web application or the store.

```python
def registration(request):
    if request.method == 'POST':
        user_form = UserSignupForm(request.POST)
        profile_form = UserProfileForm(request.POST)
        if user_form.is_valid() and profile_form.is_valid():
            user_instance = user_form.save(commit=False)
            user_instance.set_password(user_form.cleaned_data['password'])
            user_instance.save()
            profile_instance = profile_form.save(commit=False)
            profile_instance.user = user_instance
            profile_instance.save()
            return redirect('index')
    else:
        user_form = UserSignupForm()
        profile_form = UserProfileForm()
    return render(request, 'registration.html', {'user_form': user_form, 'profile_form': profile_form})
```

In the registration view function, the code handles both GET and POST requests. If the request method is POST, it processes the submitted user registration form data. It validates the form data, saves the user instance, sets the password, saves the profile instance, and redirects the user to the homepage (index). If the request method is GET, it renders the registration.html template, along with the empty user signup form (UserSignupForm) and profile form (UserProfileForm), allowing users to register.

4.Models

Django's models define the structure of the database tables and encapsulate the business logic of the application. They interact with the database through the ORM (Object-Relational Mapping), which translates model operations into SQL queries. Each model class corresponds to a database table, and fields within the model represent columns in the table. Through models, developers can define relationships between different entities, such as one-to-many and many-to-many relationships, and perform CRUD (Create, Read, Update, Delete) operations on the data stored in the database

```
from django.db import models
from django.contrib.auth.models import User
```

This part of the code imports necessary modules and resources for defining Django models, including models from django.db and User model from django.contrib.auth.models.

```
class Product(models.Model):
    name = models.CharField(max_length=200)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    image = models.ImageField(upload_to='product_images/')

    def __str__(self):
        return self.name
```

The Product model represents a product in the store. It has fields for name, category (linked to Category model through a foreign key), description, price, and image. Each product belongs to a category and has associated information like description, price, and image

```python
class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name
```

The Category model represents a category of products. It has a single field name, which is a CharField. Each category instance will have a name associated with it..

```python
class Cart(models.Model):
    user = models.ForeignKey(UserProfile, on_delete=models.CASCADE)
    products = models.ManyToManyField(Product, through='CartItem')

    def __str__(self):
        return f"Cart for {self.user}"
```

The Cart model represents a shopping cart for a user. It is linked to a UserProfile through a foreign key. It also includes a ManyToMany relationship with Product through the CartItem intermediary model.

```python
class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    address = models.CharField(max_length=255)
    phone_number = models.CharField(max_length=15)
    location = models.CharField(max_length=100)  # Add location field

    def __str__(self):
        return self.user.username
```

The UserProfile model extends the built-in User model provided by Django. It adds additional fields such as address, phone_number, and location to store user-specific information.

```python
class CartItem(models.Model):
    cart = models.ForeignKey(Cart, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return f"{self.quantity} of {self.product.name} in cart for {self.cart.user}"
```

The CartItem model represents an item in a user's shopping cart. It links a Cart and a Product, indicating the quantity of the product in the cart.

5.SETTING:

In Django, the settings file (settings.py) serves as a central configuration file for your Django project. It contains various settings that control the behavior of your Django application, including database configuration, middleware settings, static files settings, template settings, and much more. Let's explore some of the key reasons why the settings file is used and what it includes:

Configuration: The settings file allows you to configure your Django project according to your specific requirements. You can customize various aspects of your application, such as the database backend, internationalization settings, time zone settings, and logging configurations.

Database Configuration: You specify the database configuration in the settings file, including the database engine, database name, user credentials, host, and port. Django supports multiple database engines, such as PostgreSQL, MySQL, SQLite, and Oracle.

Installed Apps: Django applications are composed of reusable components called apps. In the settings file, you list the installed apps that are part of your project. Django uses this information to discover and load the necessary components for your project.

Middleware: Middleware components are reusable functions or classes that intercept HTTP requests and responses. They perform various tasks such as authentication, session management, CSRF protection, and content compression. You configure middleware classes in the MIDDLEWARE setting in the settings file.

Static Files Settings: Django allows you to serve static files such as CSS, JavaScript, and images alongside your dynamic content. You specify the directories where your static files are located and configure settings related to static file handling in the settings file.

Template Settings: Django uses templates to generate HTML dynamically. You configure settings related to template directories, loaders, context processors, and other template-related options in the settings file.

Security Settings: Django provides built-in security features to protect your web application against common security threats. You configure settings related to security measures such as CSRF protection, XSS protection, clickjacking protection, and secure HTTPS settings in the settings file.

Debugging and Development: In development mode, you typically enable debugging features such as detailed error pages, logging of SQL queries, and auto-reloading of code changes. These settings are controlled by the DEBUG setting in the settings file.

Customization and Overrides: You can create custom settings in the settings file to override default Django settings or define project-specific configurations. This allows you to tailor Django to meet the specific requirements of your project.

Overall, the settings file in Django serves as a centralized configuration hub where you define various settings to customize the behavior of your Django application. It provides flexibility, extensibility, and control over the entire project configuration, making it an essential component of Django development.

6.Admin Interface:

The admin interface is a built-in feature of Django that provides a user-friendly interface for managing site content, users, and permissions. It allows administrators and authorized users to perform CRUD (Create, Read, Update, Delete) operations on database records without writing custom views or forms.

Key Features:

Automatic Generation: The admin interface is automatically generated based on the models defined in your Django project. Each model registered with the admin site gets its own admin interface, complete with forms for creating and editing instances.

Customization: While the default admin interface is functional out of the box, Django allows for extensive customization to tailor it to your project's specific requirements. You can customize the look and feel, define custom admin actions, customize form layouts, and even create custom admin views.

Permissions and Authentication: The admin interface integrates seamlessly with Django's authentication and permissions system. You can control access to specific admin views and actions based on user permissions, ensuring that only authorized users can perform certain tasks.

Internationalization: Django's admin interface supports internationalization (i18n) out of the box, allowing you to localize admin interface text and provide translated versions of admin site content for different languages.

7.Object-Relational Mapping (ORM):

Django's ORM is a powerful feature that abstracts the interaction between your Python code and the underlying database, allowing you to work with database records using high-level Python objects rather than raw SQL queries. It simplifies database operations, promotes code readability, and reduces development time.

Key Features:

Model Definition: Django's ORM revolves around the concept of models, which are Python classes that represent database tables. Each model class corresponds to a table in the database, and each instance of the model represents a row in that table.

CRUD Operations: With Django's ORM, you can perform CRUD operations (Create, Read, Update, Delete) on database records using intuitive Python syntax. You can create new records, retrieve existing records, update records, and delete records, all using methods provided by the model class.

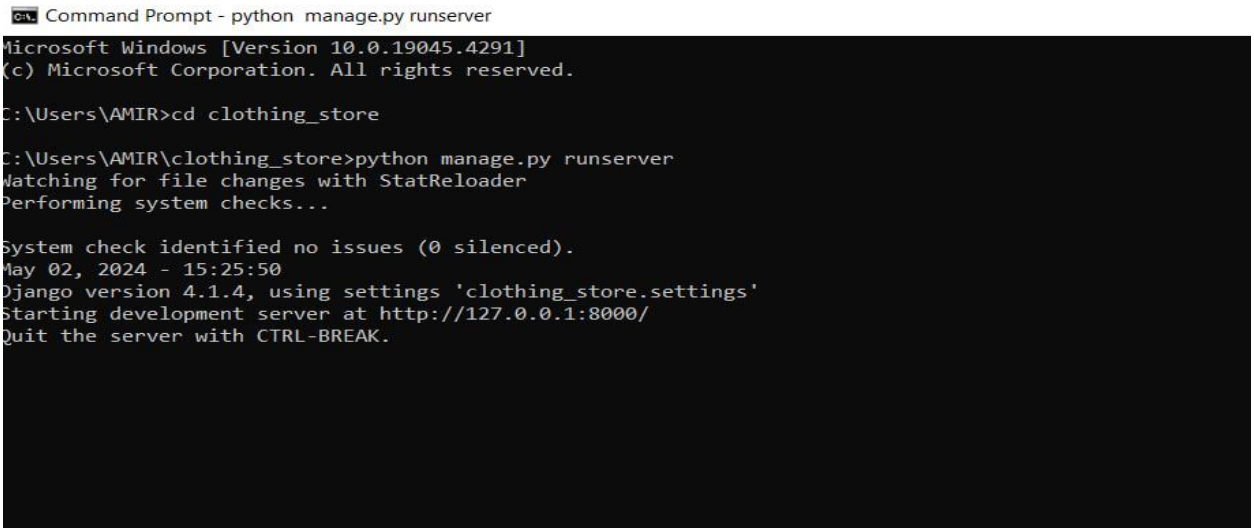
8.Querysets: Querysets are Django's abstraction for database queries. They allow you to retrieve a set of records from the database using filters, conditions, and ordering criteria. Querysets are lazy-evaluated, meaning they are executed only when needed, which improves performance.

9.Database Agnostic: Django's ORM is database-agnostic, meaning you can use it with different database backends such as PostgreSQL, MySQL, SQLite, and Oracle without changing your code. Django handles the database-specific details internally, allowing you to focus on writing Python code.

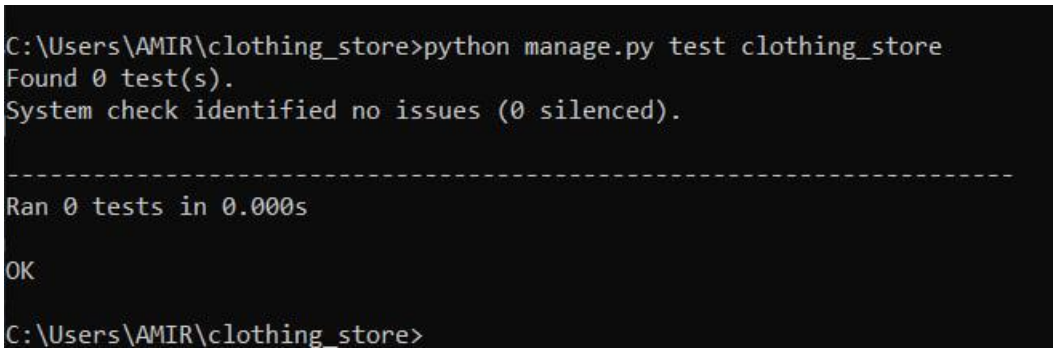
10.Automatic Migration: Django's ORM includes a built-in migration system that automates the process of synchronizing changes to your models with the underlying database schema. It generates SQL migration files based on model changes and applies them to the database, ensuring schema consistency across development, staging, and production environments.

TEST

After completing the comprehensive development process in Django, it is imperative to validate the robustness and functionality of our backend system through rigorous testing. Utilizing the command-line interface and Django's powerful management tool, namely manage.py runserver, we initiate the server to meticulously examine the backend's performance and behavior.

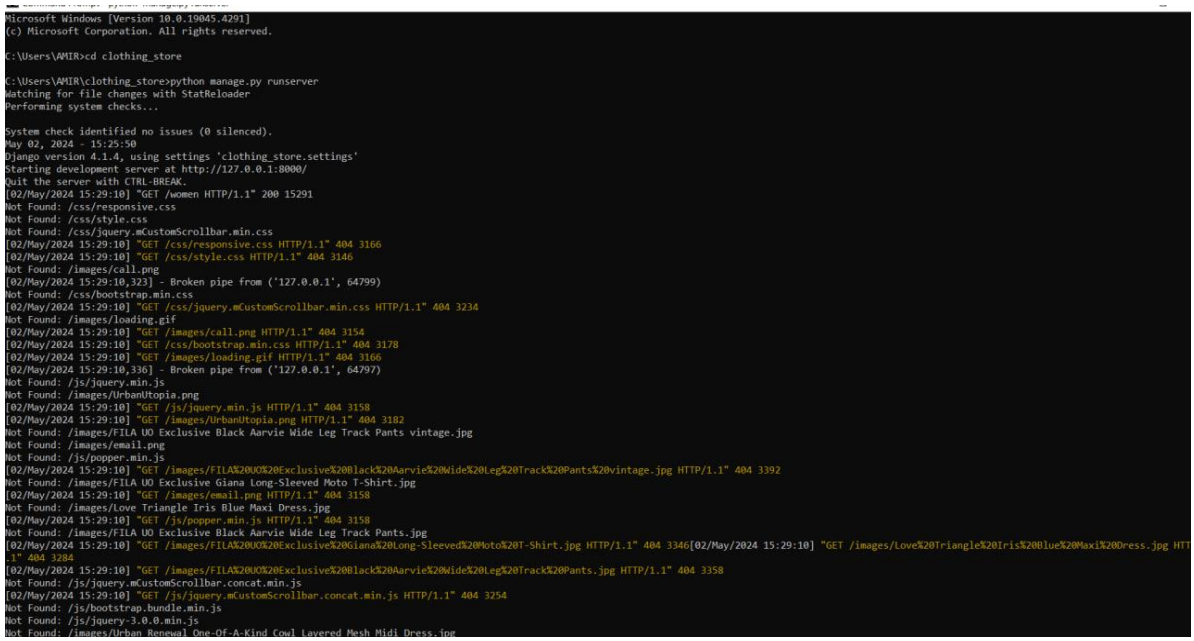


The testing phase serves as a pivotal checkpoint where we meticulously scrutinize every aspect of our backend infrastructure. As evidenced by the accompanying visual documentation, our backend system operates seamlessly without encountering any errors. However, it's crucial to acknowledge that our journey wasn't devoid of challenges. Indeed, we encountered numerous obstacles and errors initially, yet through perseverance, collaboration, and astute problem-solving, we adeptly resolved each issue.



During the testing process, our focus extended beyond mere functionality verification; we diligently assessed the scalability, efficiency, and security parameters of our backend architecture. Employing a blend of automated and manual testing methodologies, we ensured that our backend system not only meets but exceeds our stringent quality standards and user expectations.

The culmination of our endeavors has yielded a formidable and reliable backend infrastructure poised to drive our application forward. We take immense pride in the collective achievements of our team, and we eagerly anticipate showcasing our meticulously crafted product to the global audience.



Integrated development environment:

In our endeavor to cultivate an impeccable synergy between the backend and frontend components of our project, we have meticulously chosen PyCharm as our preferred Integrated Development Environment (IDE). PyCharm, developed by JetBrains, stands as a pinnacle of excellence in the realm of Python-centric software development tools, catering specifically to the discerning needs of professional developers.

Renowned for its unwavering commitment to empowering developers, PyCharm embodies a plethora of features tailored to elevate the development experience across both backend and frontend domains. With a robust suite of tools and an intuitive interface, PyCharm emerges as a stalwart companion for developers embarking on ambitious endeavors in web and scientific development.

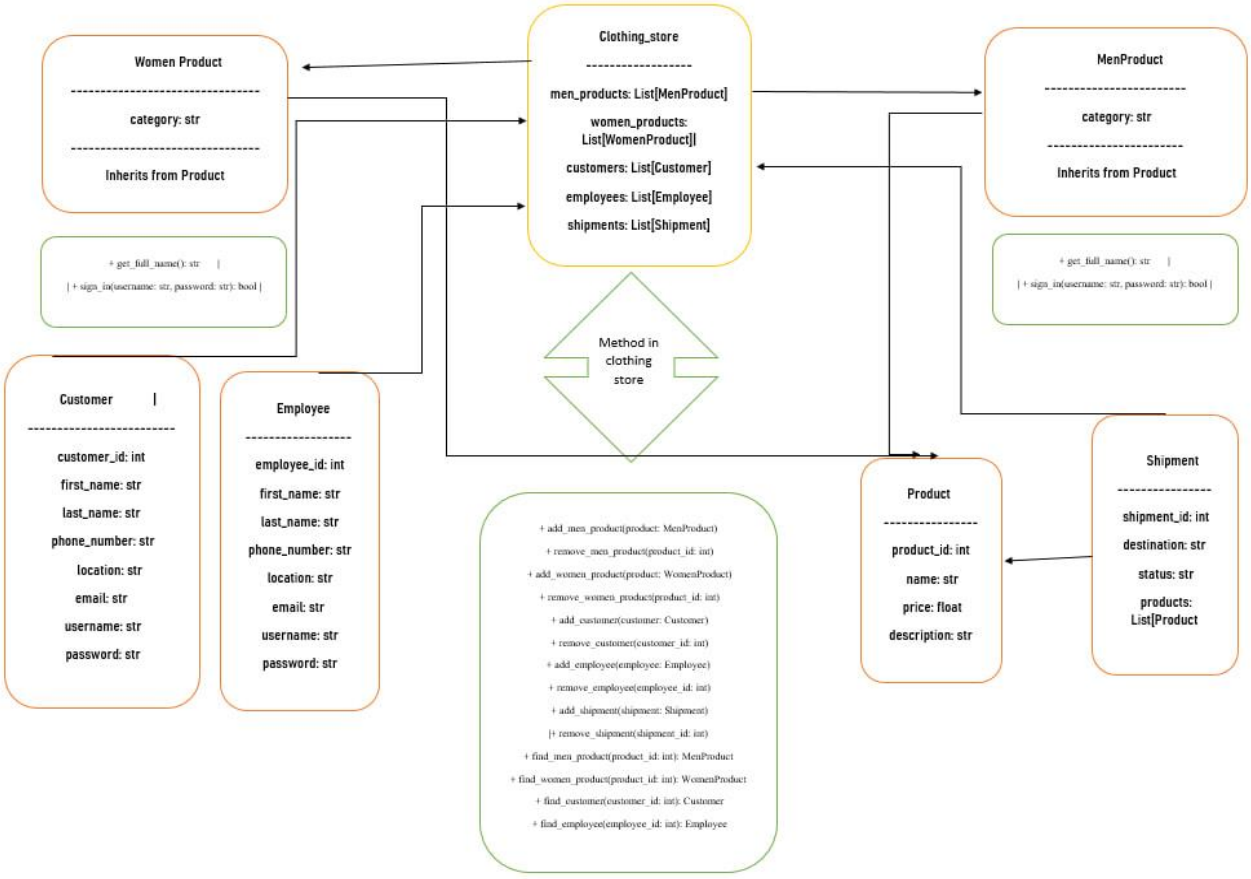
PyCharm's unparalleled versatility extends beyond conventional Python development, offering seamless support for a myriad of technologies essential for our project's success. From HTML and JavaScript for frontend endeavors to SQL for robust database

interactions, PyCharm seamlessly integrates these diverse technologies into a cohesive development environment.

Moreover, PyCharm's sophisticated code analysis capabilities, coupled with its intelligent code completion and debugging functionalities, empower developers to navigate the intricate nuances of both backend and frontend codebases with unparalleled precision and efficiency. Its comprehensive suite of plugins further enhances productivity by extending support for a wide array of frameworks, libraries, and version control systems, ensuring that our development workflow remains streamlined and agile.

By harnessing the full potential of PyCharm, we embark on a journey marked by innovation, efficiency, and excellence. With PyCharm as our trusted ally, we are equipped to surmount the most formidable challenges and deliver a product that embodies the pinnacle of sophistication and quality in both backend and frontend domains.

Class diagram



The ClothingStore class manages lists of products, including both men's and women's items. It has associations with MenProduct and WomenProduct classes through the men_products and women_products attributes. Methods in ClothingStore facilitate adding, removing, and finding products.

MenProduct and WomenProduct inherit from the Product class, representing a generalization/specialization relationship. They inherit attributes and methods from Product, enabling specific functionality tailored to men's and women's products.

Both Customer and Employee classes are associated with ClothingStore, stored within lists. They share methods for signing in, indicating a dependency on ClothingStore for authentication.

The Shipment class is linked to ClothingStore and Product. Shipment objects are stored in ClothingStore's list, and it contains a list of products being shipped.