NIIT

final Project (Ecommerce Website) Documentation – Mr. David TEMITAYO Adediji

# Project Details

|  |  |
| --- | --- |
| Project Name | Coding Course Ordering Ecommerce Website |
| Student Name | Mr. David Adediji |
| Name of Instructor | Mr. Fredrick Jones |
| Presentation Date | 2nd of June, 2023 |
| Program Name | MMS |
|  |  |

# Acknowledgements

I want to deeply express gratitude to my Program Instructors, Mr. Belba Ngoy, Mr. Efe Otega & Ms. Esther Iyege, Mr. Fredrick Jones, for taking the class diligently through my courses. Their lessons were very concise and easy to understand. I also appreciate the opportunity given to defend a project pertaining to what I had been thought and had understood. More had been possible with the assistance of my helpful colleagues who helped make the lessons interactive, fun and had even contributed indirectly to the development of my Ecommerce Project.

# Project Overview

* The name of the ecommerce website is Code Avenue.
* This web project using jQuery and Django was created to enable users to buy programming courses.
* This project idea is inspired by Code with Mosh’s approach to ecommerce for his programming courses.
* The web application was created using HTML, CSS, jQuery programming language and Django REST API framework, stripe alongside other resources.

# Project Features

# Ecommerce Logic to Allow Users to Buy Courses: Implement a robust ecommerce system that enables users to browse through a catalog of courses and make purchases. Provide clear course descriptions, pricing information, and a seamless checkout process. Allow users to add courses to their shopping cart, view their cart contents, and proceed to secure payment.

# Ecommerce Logic to Allow Users to Subscribe for Courses: In addition to individual course purchases, offer users the option to subscribe to courses. Implement a subscription-based model that provides users with access to a curated selection of courses for a recurring fee. Allow users to manage their subscriptions, view subscription details, and easily cancel or modify their subscription plans.

# Payment Using Stripe API: Integrate the Stripe API to handle secure and reliable payment processing. Enable users to enter their payment information, such as credit card details, and process transactions securely. Implement error handling and provide users with confirmation of successful payments.

# Blog to Convey Information about Programming: Incorporate a blog section into the website to publish informative and engaging articles about programming. Cover various programming languages, frameworks, best practices, and tutorials. Allow users to comment on blog posts and share them on social media platforms.

# Profile Page to Manage User Information: Create a user profile page where users can manage their personal information, such as name, email address, and profile picture. Implement functionality for users to update their details, change passwords, and view their purchase history or subscription status.

# Properly Validated Logic and Signup Page: Ensure that the logic and input on the website are properly validated to prevent unauthorized access and enhance security. Implement form validation on the signup page to ensure accurate user registration and authentication. Use techniques such as CAPTCHA to deter bots and spam registrations.

# Informative Learning Paths to Direct People in Programming Education: Design and implement informative learning paths or roadmaps that guide users through their programming education journey. Provide clear pathways based on skill levels or specific programming goals, suggesting suitable courses to take at each stage. Allow users to track their progress and mark completed milestones.

# Instructor Accessible via Email Messaging on Contact Page: Create a contact page that allows users to reach out to instructors or support staff via email messaging. Implement a form where users can submit their queries or feedback, and ensure prompt responses from the appropriate personnel.

# Beautiful Lollipop Design: Craft an aesthetically pleasing and user-friendly design for the website, incorporating a lollipop-inspired visual theme. Employ attractive colors, typography, and intuitive navigation to enhance the overall user experience. Ensure the design is responsive and accessible across different devices and screen sizes.

# User Authentication Using Django: Utilize the Django framework's built-in authentication system to handle user registration, login, and session management. Implement secure password hashing, password reset functionality, and user permissions to control access to certain features or content.

# Project Hosted on Heroku and GitHub Pages: Deploy the web project on Heroku to ensure it is accessible to users online. Configure the necessary server settings and dependencies. Additionally, consider using GitHub Pages to host static content, such as the blog, to enhance performance and availability

# Project Process

* Design of wireframe using Figma
* Design of flowcharts for program logic
* Creation of Website markup with HTML
* Styling of website with CSS
* Programming of ecommerce website Logic with jQuery
* Repeating process to build peripheral blog
* Built backend with Django
* Payment Processing with Stripe
* Testing
* Live Hosting

# Technical Lessons applied in project

* Ajax Requests: In the web project, Ajax requests were utilized to enhance the user experience by allowing asynchronous communication with the server. Ajax enables sending and receiving data from the server without reloading the entire webpage, resulting in a smoother and more responsive user interface.
* Load Function: The Load function, commonly used in jQuery, was employed to dynamically load content into specific sections of the webpage. This technique helps improve performance by fetching only the necessary data or HTML elements when required, rather than loading everything at once.
* jQuery Selectors: jQuery selectors were employed to efficiently target and manipulate HTML elements on the webpage. By using CSS-like selectors, jQuery simplifies the process of identifying and modifying specific elements, such as applying styles or binding event handlers.
* Local Storage: The Local Storage API was utilized to store and retrieve data on the client-side. This feature allowed the web project to save user preferences, session information, or temporary data locally within the user's browser, reducing the need for frequent server requests.
* jQuery Functions: Various jQuery functions were utilized to perform common tasks and enhance interactivity within the web project. These functions include animations, DOM manipulation, form validation, and handling of events, among others.
* String Literals: String literals were used to construct dynamic strings in the web project. By incorporating variables or dynamically generated values into string literals, it becomes easier to generate customized content or construct URLs for Ajax requests.
* jQuery Event Listeners: jQuery event listeners were applied to detect and respond to user interactions, such as clicks, form submissions, or keyboard input. These event listeners allow for the execution of specific actions or the modification of the webpage in response to user events.
* Django ORM: The Django ORM (Object-Relational Mapping) was utilized in the project to interact with the database. By using Django's ORM, developers can work with the database using Python objects and methods, making database operations more intuitive and efficient.
* Stripe API: The Stripe API was integrated into the project to handle online payments securely. By utilizing the Stripe API, the web project was able to process credit card payments, manage subscriptions, and handle other financial transactions seamlessly.
* Django Rest Framework: The Django Rest Framework (DRF) was employed to build robust and scalable APIs (Application Programming Interfaces) in the web project. DRF provides a set of tools and libraries to simplify API development, authentication, serialization, and handling of HTTP requests.
* Django Authentication: Django's built-in authentication system was utilized to handle user authentication and authorization in the web project. This feature allowed users to create accounts, log in, log out, and access specific functionalities based on their permissions.
* Atypical CSS Selectors: The project employed atypical CSS selectors to target specific HTML elements based on non-standard criteria. These selectors allow for more precise and specialized styling or manipulation of elements that may not be easily achieved using standard CSS selectors.
* AI Video Generation: The web project incorporated AI video generation techniques to automate the process of creating videos. By leveraging AI algorithms, the project was able to generate dynamic and personalized video content based on user inputs or predefined criteria.
* Git Version Control: Git version control was employed to manage and track changes to the web project's source code. By using Git, developers could collaborate, maintain a history of changes, and easily revert to previous versions if needed, ensuring better code management and project stability.
* These technical lessons applied in your web project demonstrate a diverse range of skills and technologies, enabling the development of a responsive, interactive, and feature-rich web application.

# Django Rest Framework

Django Rest Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django, a popular Python web framework. It provides a set of tools and libraries that simplify the process of building APIs, handling requests, serialization, authentication, and more. DRF is designed to promote best practices and enables developers to create scalable and secure RESTful APIs with minimal code.

# Django Model, View & Serializer

Model:

In web development, a model represents the data structure and the logic behind it. It defines how data is stored, organized, and manipulated within a web application. Models typically correspond to database tables or collections and provide an abstraction layer for interacting with the data.

Views:

Views are responsible for handling HTTP requests and generating HTTP responses in a web application. They receive requests from users, process the data, interact with models or services, and render the appropriate response, such as HTML templates or JSON data. Views play a crucial role in determining what content should be presented to the user based on their request.

URLs:

URLs (Uniform Resource Locators) are the web addresses that users access to interact with specific resources or pages on a website. In web development frameworks, URLs are mapped to views, determining which view should handle a particular URL pattern. URL patterns are defined in a project's URL configuration, allowing for a clean and organized way of routing requests to the appropriate views.

Serializers:

Serializers are components used in web development frameworks to convert complex data, such as database models or querysets, into formats that can be easily transmitted and rendered, such as JSON or XML. They provide a way to serialize or deserialize data when communicating between the client and server. Serializers also handle data validation, making sure the received data is in the expected format before further processing.

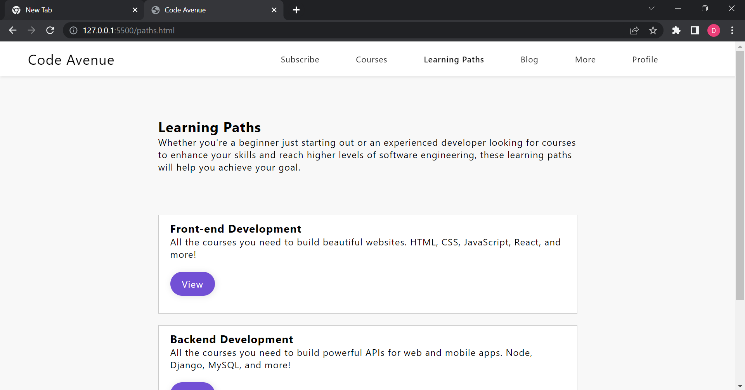
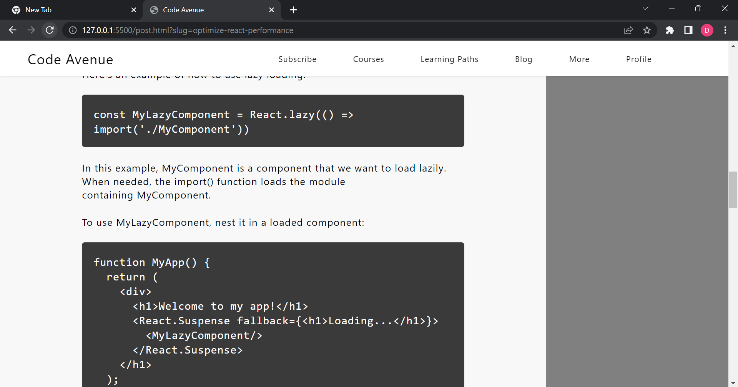
Overall, models, views, URLs, and serializers are key components of web development frameworks. They work together to define the data structure, handle user requests, map URLs to appropriate views, and serialize/deserialize data for communication between the client and server. Understanding these concepts is essential for building robust and scalable web applications.

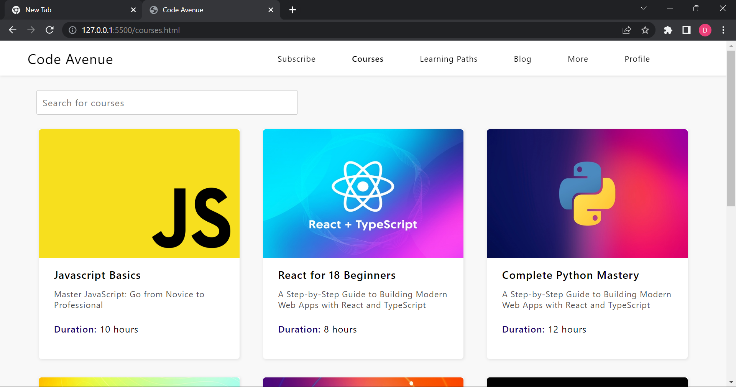
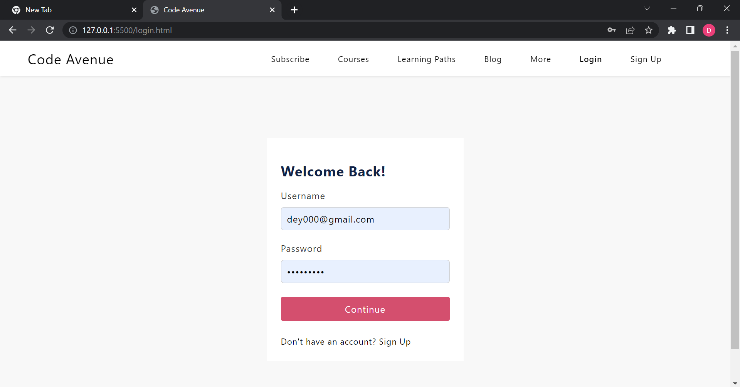
# Project Images

# 

# 

# 

# Django Models

core/models.py:

from django.contrib.auth.models import AbstractUser

from django.db import models

# Create your models here.

class User(AbstractUser):

email = models.EmailField(unique=True)

class ContactMessage(models.Model):

email = models.EmailField()

message = models.TextField()

timestamp = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return self.email

blog/models.py:

from django.db import models

from django.conf import settings

from ckeditor.fields import RichTextField

class Category(models.Model):

name = models.CharField(max\_length=100)

slug = models.SlugField(max\_length=100, unique=True)

def \_\_str\_\_(self):

return self.name

class Post(models.Model):

title = models.CharField(max\_length=255)

slug = models.SlugField(max\_length=255, unique=True)

author = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

content = RichTextField()

created\_on = models.DateTimeField(auto\_now\_add=True)

updated\_on = models.DateTimeField(auto\_now=True)

category = models.ForeignKey(Category, on\_delete=models.CASCADE)

def \_\_str\_\_(self):

return self.title

class Comment(models.Model):

post = models.ForeignKey(Post, on\_delete=models.CASCADE, related\_name='comments')

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

email = models.EmailField()

content = models.TextField()

created\_on = models.DateTimeField(auto\_now\_add=True)

approved = models.BooleanField(default=False)

class Meta:

unique\_together = (('post', 'user'),)

def \_\_str\_\_(self):

return self.content

class Subscriber(models.Model):

email = models.EmailField(unique=True)

subscribed\_on = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return self.email

course\_store/models.py:  
from django.db import models

from django.conf import settings

from django.utils import timezone

# Course model class to represent a course

class Course(models.Model):

LEVEL\_CHOICES = (

('beginner', 'Beginner'),

('intermediate', 'Intermediate'),

('advanced', 'Advanced'),

)

level = models.CharField(max\_length=20, choices=LEVEL\_CHOICES)

endpoints = models.TextField()

features = models.TextField()

name = models.CharField(max\_length=35)

description = models.TextField()

summary = models.CharField(max\_length=80)

image = models.ImageField(upload\_to='course\_images/', default="./course\_images/js.png")

time\_length = models.PositiveSmallIntegerField()

price = models.DecimalField(max\_digits=10, decimal\_places=2)

discount\_percentage = models.PositiveSmallIntegerField()

tagline = models.CharField(max\_length=55)

available = models.BooleanField(default=True)

downloadable\_file = models.FileField(upload\_to='course\_files/', blank=True)

shortform = models.CharField(max\_length=35)

def \_\_str\_\_(self):

return self.name

class BillingAddress(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

street\_address = models.CharField(max\_length=255)

city = models.CharField(max\_length=255)

country = models.CharField(max\_length=255)

postal\_code = models.CharField(max\_length=255)

def \_\_str\_\_(self):

return f'Billing address for {self.user.username}'

# Order model class to represent an order placed by a user for a course

class Order(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

course = models.ForeignKey(Course, on\_delete=models.CASCADE)

price = models.DecimalField(max\_digits=10, decimal\_places=2)

date\_ordered = models.DateTimeField(auto\_now\_add=True)

billing\_address = models.ForeignKey(BillingAddress, on\_delete=models.PROTECT)

downloaded = models.BooleanField(default=False)

def \_\_str\_\_(self):

return f'{self.user.username} ordered {self.course.name} on {self.date\_ordered}'

# Payment model class to represent a payment made by a user for an order

class Payment(models.Model):

order = models.OneToOneField(Order, on\_delete=models.CASCADE)

amount = models.DecimalField(max\_digits=10, decimal\_places=2)

card\_number = models.BigIntegerField()

card\_expiry\_date = models.DateField()

cvc = models.IntegerField()

intent\_id = models.CharField(max\_length=255)

card\_holder = models.CharField(max\_length=255)

def \_\_str\_\_(self):

return f'{self.order.user.username} paid {self.amount} for {self.order.course.name}'

# Review model class to represent a review made by a user for a course

class Review(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

course = models.ForeignKey(Course, on\_delete=models.CASCADE)

content = models.TextField()

rating = models.IntegerField()

def \_\_str\_\_(self):

return f'{self.user.username} reviewed {self.course.name}'

from django.db.models.signals import post\_save

from django.dispatch import receiver

from datetime import date

# Order model class to represent an order placed by a user for a course

class SubscriptionOrder(models.Model):

DURATION\_CHOICES = (

('monthly', 'Monthly'),

('annual', 'Annual'),

)

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

duration = models.CharField(max\_length=10, choices=DURATION\_CHOICES)

date\_ordered = models.DateTimeField(auto\_now\_add=True)

billing\_address = models.ForeignKey(BillingAddress, on\_delete=models.PROTECT)

def \_\_str\_\_(self):

return f'{self.user.username} ordered for all access on {self.date\_ordered}'

class SubscriptionPayment(models.Model):

order = models.OneToOneField(SubscriptionOrder, on\_delete=models.CASCADE)

amount = models.DecimalField(max\_digits=10, decimal\_places=2)

card\_number = models.BigIntegerField()

card\_expiry\_date = models.DateField()

cvc = models.IntegerField()

intent\_id = models.CharField(max\_length=255)

card\_holder = models.CharField(max\_length=255)

def \_\_str\_\_(self):

return f'{self.order.user.username} paid {self.amount} for all access'

class AllAccessSubscriber(models.Model):

user = models.ForeignKey(settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE)

subscription\_start\_date = models.DateField(auto\_now\_add=True)

subscription\_end\_date = models.DateField()

is\_active = models.BooleanField(default=True)

subscription\_payment = models.OneToOneField(SubscriptionPayment, on\_delete=models.CASCADE)

# Other fields and methods specific to the AllAccessSubscriber model

@receiver(post\_save, sender=AllAccessSubscriber)

def update\_subscription\_status(sender, instance, \*\*kwargs):

if instance.subscription\_end\_date < date.today():

instance.is\_active = False

instance.save()

## Django views

Blog/views.py

from rest\_framework import generics, filters, permissions, status

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework.pagination import PageNumberPagination

from django.shortcuts import get\_object\_or\_404

from .models import Post, Category, Comment

from .serializers import PostSerializer, CategorySerializer, CommentSerializer

class PostPagination(PageNumberPagination):

page\_size = 5 # Change this to set the number of posts to display per page

page\_size\_query\_param = 'limit'

max\_page\_size = 100

class PostList(generics.ListAPIView):

queryset = Post.objects.all()

serializer\_class = PostSerializer

pagination\_class = PostPagination

permission\_classes = [permissions.AllowAny]

class PostByCategoryAPIView(generics.ListAPIView):

serializer\_class = PostSerializer

filter\_backends = [filters.SearchFilter]

search\_fields = ['category\_\_name']

permission\_classes = [permissions.AllowAny]

pagination\_class = PostPagination

def get\_queryset(self):

category\_slug = self.kwargs['category\_slug']

category = get\_object\_or\_404(Category, slug=category\_slug)

return Post.objects.filter(category=category)

class PostDetail(generics.RetrieveAPIView):

queryset = Post.objects.all()

serializer\_class = PostSerializer

lookup\_field = 'slug'

permission\_classes = [permissions.AllowAny]

class CategoryList(generics.ListAPIView):

queryset = Category.objects.all()

serializer\_class = CategorySerializer

permission\_classes = [permissions.AllowAny]

@api\_view(['POST'])

def create\_comment(request):

post\_id = request.data.get('postId')

content = request.data.get('content')

user = request.user

try:

post = Post.objects.get(id=post\_id)

except Post.DoesNotExist:

return Response({'error': 'Post not found'}, status=status.HTTP\_404\_NOT\_FOUND)

serializer = CommentSerializer(data={'post': post.id, 'user': user.id, 'email': user.email, 'content': content}) serializer.is\_valid(raise\_exception=True) serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

class DeleteComment(generics.DestroyAPIView):

queryset = Comment.objects.all()

serializer\_class = CommentSerializer

permission\_classes = [permissions.IsAuthenticated]

def get\_object(self):

comment = get\_object\_or\_404(Comment, id=self.kwargs.get('pk'))

if comment.user != self.request.user:

raise permissions.PermissionDenied

return comment

def delete(self, request, \*args, \*\*kwargs):

return self.destroy(request, \*args, \*\*kwargs)

Core/views.py

from django.contrib.auth import authenticate, login

from django.contrib.auth.models import User

from django.views.decorators.csrf import csrf\_exempt

from django.http import JsonResponse

from rest\_framework.authtoken.models import Token

from rest\_framework.authentication import TokenAuthentication

from rest\_framework.decorators import api\_view, authentication\_classes

from rest\_framework.response import Response

from rest\_framework import status

from django.contrib.auth.models import User

from rest\_framework import generics

from rest\_framework.response import Response

from .models import ContactMessage

from .serializers import ContactMessageSerializer

class ContactMessageCreateView(generics.CreateAPIView):

queryset = ContactMessage.objects.all()

serializer\_class = ContactMessageSerializer

def post(self, request, \*args, \*\*kwargs):

serializer = self.get\_serializer(data=request.data) serializer.is\_valid(raise\_exception=True) serializer.save()

return Response(serializer.data, status=201)

@api\_view(['GET'])

@authentication\_classes([TokenAuthentication])

def get\_user\_id(request):

user = request.user

user\_id = user.id

# user\_id = user.email

return Response({'user\_id': user\_id}, status=status.HTTP\_200\_OK)

@csrf\_exempt

def login\_view(request):

if request.method == 'POST':

username = request.POST.get('username')

password = request.POST.get('password')

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

# print(user.is\_authenticated)

if user is not None:

login(request, user)

token, created = Token.objects.get\_or\_create(user=user) return JsonResponse({'token': token.key})

else:

return JsonResponse({'message': 'Invalid username or password.'}, status=400)

else:

return JsonResponse({'message': 'Invalid request method.'}, status=400)

@csrf\_exempt

def signup(request):

# print(request.method)

if request.method == 'POST':

email = request.POST.get('email')

password = request.POST.get('password')

first\_name = request.POST.get('first\_name')

last\_name = request.POST.get('last\_name')

username = request.POST.get('username')

# Validate the input

if not email or not password or not first\_name or not last\_name:

return JsonResponse({'error': 'All fields are required'}, status=400)

# Check if the email is already taken if User.objects.filter(email=email).exists():

return JsonResponse({'error': 'This email is already taken'}, status=400)

# Create the new user

user = User.objects.create\_user(username=username, email=email, password=password)

user.first\_name = first\_name

user.last\_name = last\_name

user.save()

# Log in the new user

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

if user is not None:

login(request, user)

return JsonResponse({'success': 'User created successfully'})

else:

return JsonResponse({'error': 'Could not log in the new user'}, status=400)

else:

return JsonResponse({'error': 'Invalid request method'}, status=400)

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import status

from django.contrib.auth import get\_user\_model

from .serializers import UserSerializer

User = get\_user\_model()

class UserUpdateView(APIView):

def put(self, request, \*args, \*\*kwargs):

user = request.user

serializer = UserSerializer(user, data=request.data, partial=True)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_200\_OK)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

from django.contrib.auth import get\_user\_model

from rest\_framework import generics

from rest\_framework.response import Response

from rest\_framework import status

User = get\_user\_model()

# class UserDetailView(generics.RetrieveUpdateDestroyAPIView):

# queryset = User.objects.all()

# serializer\_class = UserSerializer

# def get(self, request, \*args, \*\*kwargs):

# user\_id = kwargs['user\_id']

# try:

# user = User.objects.get(id=user\_id)

# except User.DoesNotExist:

# return Response(status=status.HTTP\_404\_NOT\_FOUND)

# serializer = UserSerializer(user)

# return Response(serializer.data, status=status.HTTP\_200\_OK)

# def delete(self, request, \*args, \*\*kwargs):

# user\_id = kwargs['user\_id']

# try:

# user = User.objects.get(id=user\_id)

# except User.DoesNotExist:

# return Response(status=status.HTTP\_404\_NOT\_FOUND)

# user.delete()

# return Response(status=status.HTTP\_204\_NO\_CONTENT)

class UserDetailView(generics.RetrieveUpdateDestroyAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

def get(self, request, \*args, \*\*kwargs):

user = request.user

serializer = UserSerializer(user)

return Response(serializer.data, status=status.HTTP\_200\_OK)

def delete(self, request, \*args, \*\*kwargs):

user = request.user

user.delete()

return Response(status=status.HTTP\_204\_NO\_CONTENT)

def put(self, request, \*args, \*\*kwargs):

user = request.user

serializer = UserSerializer(user, data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_200\_OK)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

Course\_store/view.py

from django.db.models import Count

from django.conf import settings

from decimal import Decimal

from rest\_framework import generics, permissions

from rest\_framework.response import Response

from rest\_framework.pagination import PageNumberPagination

from .models import Course, Order, Payment, Review, BillingAddress, SubscriptionOrder, SubscriptionPayment

from .serializers import (

CourseSerializer,

OrderSerializer,

PaymentSerializer,

ReviewSerializer,

BillingAddressSerializer,

SubscriptionOrderSerializer,

SubscriptionPaymentSerializer

)

class CoursePagination(PageNumberPagination):

page\_size = 9 # Change this to set the number of posts to display per page

page\_size\_query\_param = 'limit'

max\_page\_size = 100

class CourseList(generics.ListAPIView):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

permission\_classes = [permissions.AllowAny]

# pagination\_class = CoursePagination

class CourseDetail(generics.RetrieveAPIView):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

permission\_classes = [permissions.AllowAny]

class TopSellingCoursesView(generics.ListAPIView):

queryset = Course.objects.annotate(num\_orders=Count('order')).order\_by('-num\_orders')[:6]

serializer\_class = CourseSerializer

permission\_classes = [permissions.AllowAny]

class BillingAddressCreateView(generics.CreateAPIView):

model = BillingAddress

serializer\_class = BillingAddressSerializer

def perform\_create(self, serializer):

print(self.request.user)

serializer.save(user=self.request.user)

class BillingAddressRetrieveView(generics.RetrieveAPIView):

serializer\_class = BillingAddressSerializer

def get(self, request, \*args, \*\*kwargs):

user = request.user

try:

billing\_address = BillingAddress.objects.get(user=user)

except BillingAddress.DoesNotExist:

return Response({'error': 'Billing address not found.'}, status=status.HTTP\_404\_NOT\_FOUND)

serializer = self.serializer\_class(billing\_address)

return Response(serializer.data, status=status.HTTP\_200\_OK)

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework import status

@api\_view(['POST'])

def create\_billing\_address(request):

street\_address = request.data.get('street\_address')

city = request.data.get('city')

country = request.data.get('country')

postal\_code = request.data.get('postal\_code')

user = request.user

# Check if the user already has a billing address

try:

billing\_address = BillingAddress.objects.get(user=user)

except BillingAddress.DoesNotExist:

billing\_address = None

# Save the billing address object to the database

data = {

'user': user.id,

'street\_address': street\_address,

'city': city,

'country': country,

'postal\_code': postal\_code,

}

serializer = BillingAddressSerializer(instance=billing\_address, data=data)

serializer.is\_valid(raise\_exception=True)

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import status

from .serializers import BillingAddressSerializer

class BillingAddressUpdateView(APIView):

serializer\_class = BillingAddressSerializer

def put(self, request, \*args, \*\*kwargs):

user = request.user

try:

billing\_address = BillingAddress.objects.get(user=user)

except BillingAddress.DoesNotExist:

return Response({'error': 'Billing address not found.'}, status=status.HTTP\_404\_NOT\_FOUND)

serializer = self.serializer\_class(billing\_address, data=request.data, partial=True)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_200\_OK)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

# class OrderList(generics.ListCreateAPIView):

# queryset = Order.objects.all()

# serializer\_class = OrderSerializer

# permission\_classes = [permissions.IsAuthenticated]

# def perform\_create(self, serializer):

# serializer.save(user=self.request.user)

# return serializer.instance

@api\_view(['POST'])

def create\_order(request):

course\_id = request.data.get('course\_id')

user = request.user

ba = request.data.get('ba')

# Retrieve the course object from the database

try:

course = Course.objects.get(id=course\_id)

ba = BillingAddress.objects.get(id=ba)

except Course.DoesNotExist:

return Response({'error': 'Course not found'}, status=status.HTTP\_404\_NOT\_FOUND)

# Calculate the price of the course

from decimal import Decimal, ROUND\_HALF\_UP

# Calculate the price of the course

price = Decimal(str(course.price)) \* Decimal(((100-course.discount\_percentage)/100))

# print(price)

price = price.quantize(Decimal('.01'), rounding=ROUND\_HALF\_UP)

# Check if the user already has an order for this course

# try:

# order = Order.objects.get(user=user, course=course)

# except Order.DoesNotExist:

# order = None

# Save the order object to the database

data = {

'user': user.id,

'course': course.id,

'price': price,

'billing\_address': ba.id

}

serializer = OrderSerializer(data=data)

try:

serializer.is\_valid(raise\_exception=True)

except Exception as e:

print(e)

return Response({'error': 'Validation error'}, status=status.HTTP\_400\_BAD\_REQUEST)

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

from django.conf import settings

@api\_view(['POST'])

def create\_subscription\_order(request):

user = request.user

duration = request.data.get("duration")

ba = request.data.get('ba')

# Retrieve the course object from the database

try:

ba = BillingAddress.objects.get(id=ba)

except BillingAddress.DoesNotExist:

return Response({'error': 'Billing Address not found'}, status=status.HTTP\_404\_NOT\_FOUND)

from decimal import Decimal, ROUND\_HALF\_UP

subscription\_price\_monthly = settings.SUBSCRIPTION\_PRICE\_MONTHLY

subscription\_price\_monthly\_discount = settings.SUBSCRIPTION\_PRICE\_MONTHLY\_DISCOUNT

subscription\_price\_yearly = settings.SUBSCRIPTION\_PRICE\_YEARLY

subscription\_price\_yearly\_discount = settings.SUBSCRIPTION\_PRICE\_YEARLY\_DISCOUNT

print(duration)

if duration == 'monthly':

price = Decimal(str(subscription\_price\_monthly)) \* Decimal(subscription\_price\_monthly\_discount / 100)

else:

price = Decimal(str(subscription\_price\_yearly)) \* Decimal(subscription\_price\_yearly\_discount / 100)

price = price.quantize(Decimal('0.00'), rounding=ROUND\_HALF\_UP)

# Check if the user already has an order for this course

# try:

# order = Order.objects.get(user=user, course=course)

# except Order.DoesNotExist:

# order = None

# Save the order object to the database

data = {

'user': user.id,

'duration': duration,

'price': price,

'billing\_address': ba.id

}

serializer = SubscriptionOrderSerializer(data=data)

try:

serializer.is\_valid(raise\_exception=True)

except Exception as e:

print(e)

return Response({'error': 'Validation error'}, status=status.HTTP\_400\_BAD\_REQUEST)

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

class OrderDetail(generics.RetrieveAPIView):

queryset = Order.objects.all()

serializer\_class = OrderSerializer

permission\_classes = [permissions.IsAuthenticated]

@api\_view(['POST'])

def payment\_list(request):

try:

order\_id = int(request.data.get('order'))

except (TypeError, ValueError):

return Response({'error': 'Invalid order ID'}, status=status.HTTP\_400\_BAD\_REQUEST)

try:

order = Order.objects.get(id=order\_id)

except Order.DoesNotExist:

return Response({'error': 'Order not found'}, status=status.HTTP\_404\_NOT\_FOUND)

amount = int(order.price \* 100)

order\_serializer = OrderSerializer(order)

data = {

'order': order\_serializer.data,

'amount': amount,

'card\_holder': request.data.get('card\_holder'),

'card\_number': request.data.get('card\_number'),

'card\_expiry\_date': request.data.get('card\_expiry\_date'),

'cvc': request.data.get('cvc'),

}

serializer = PaymentSerializer(data=data)

if serializer.is\_valid():

print(data)

# Create the Stripe payment intent

stripe.api\_key = settings.STRIPE\_SECRET\_KEY

try:

intent = stripe.PaymentIntent.create(

amount=amount,

currency='usd',

description=f'Payment for order #{order.id}',

payment\_method\_types=['card'],

payment\_method\_data={

'type': 'card',

'card': {

'token': 'tok\_visa', # Replace with the appropriate test token

},

}

)

except stripe.error.CardError as e:

# Since it's a decline, stripe.error.CardError will be caught

body = e.json\_body

err = body.get('error', {})

print(body) # Add this line to print the response from Stripe

return Response({'error': f"Card declined: {err.get('message')}"}, status=status.HTTP\_400\_BAD\_REQUEST)

# Save the payment intent ID in the Payment model

payment = serializer.save(order=order)

payment.intent\_id = intent.id

payment.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

else:

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

from rest\_framework import status

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from .models import AllAccessSubscriber

import stripe

from datetime import date, timedelta

@api\_view(['POST'])

def subscription\_payment\_list(request):

try:

order\_id = int(request.data.get('order'))

except (TypeError, ValueError):

return Response({'error': 'Invalid order ID'}, status=status.HTTP\_400\_BAD\_REQUEST)

try:

order = SubscriptionOrder.objects.get(id=order\_id)

except Order.DoesNotExist:

return Response({'error': 'Order not found'}, status=status.HTTP\_404\_NOT\_FOUND)

if order.duration == "annual":

amount = settings.SUBSCRIPTION\_PRICE\_YEARLY

price = "price\_1NDGbKEDNiG9zz7soWKcb0eA"

elif order.duration == "monthly":

amount = settings.SUBSCRIPTION\_PRICE\_MONTHLY

price = "price\_1NDWqSEDNiG9zz7sWPGR3lKi"

order\_serializer = SubscriptionOrderSerializer(order)

data = {

'order': order\_serializer.data,

'amount': amount,

'card\_holder': request.data.get('card\_holder'),

'card\_number': request.data.get('card\_number'),

'card\_expiry\_date': request.data.get('card\_expiry\_date'),

'cvc': request.data.get('cvc'),

'payment\_method\_token': request.data.get('payment\_method\_token'),

}

serializer = SubscriptionPaymentSerializer(data=data)

if serializer.is\_valid():

# Get the user from the request object

user = request.user

# Create the Stripe payment intent

stripe.api\_key = settings.STRIPE\_SECRET\_KEY

# Create the Stripe customer with the user's email address

try:

customer = stripe.Customer.create(

email=user.email,

payment\_method=data['payment\_method\_token'], # Attach payment method to customer

invoice\_settings={

'default\_payment\_method': data['payment\_method\_token'] # Set default payment method for future invoices

}

)

except stripe.error.StripeError as e:

# Handle Stripe customer creation errors

print(e)

return Response({'error': 'Failed to create customer'}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

try:

intent = stripe.PaymentIntent.create(

amount=int(amount \* 100),

currency='usd',

description=f'Payment for order #{order.id}',

customer=customer.id, # Set customer for the payment intent

payment\_method=data['payment\_method\_token'],

)

except stripe.error.CardError as e:

# Since it's a decline, stripe.error.CardError will be caught

body = e.json\_body

err = body.get('error', {})

print(body) # Add this line to print the response from Stripe

return Response({'error': f"Card declined: {err.get('message')}"}, status=status.HTTP\_400\_BAD\_REQUEST)

# Create the Stripe subscription

try:

subscription = stripe.Subscription.create(

customer=customer.id,

items=[

{

'price': price, # Replace with your actual price ID

},

],

payment\_behavior='default\_incomplete',

expand=['latest\_invoice.payment\_intent'],

)

except stripe.error.StripeError as e:

# Handle Stripe subscription creation errors

print(e)

return Response({'error': 'Failed to create subscription'}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

# Save the payment intent ID, subscription ID, and customer ID in the SubscriptionPayment model

subscription\_payment = serializer.save(order=order)

subscription\_payment.intent\_id = intent.id

subscription\_payment.subscription\_id = subscription.id

subscription\_payment.customer\_id = customer.id

subscription\_payment.save()

# Create or update the AllAccessSubscriber for the user

subscription\_start\_date = date.today() # Current date

if order.duration == "annual":

subscription\_end\_date = subscription\_start\_date + timedelta(days=365)

else:

subscription\_end\_date = subscription\_start\_date + timedelta(days=30)

# Create a new AllAccessSubscriber for the user

all\_access\_subscriber = AllAccessSubscriber.objects.create(

user=user,

subscription\_start\_date=subscription\_start\_date,

subscription\_end\_date=subscription\_end\_date,

is\_active=True,

subscription\_payment=subscription\_payment

)

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

else:

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

from django.http import FileResponse, Http404

from django.conf import settings

import mimetypes

import os

def download\_course(request, payment\_id):

try:

payment = Payment.objects.get(id=payment\_id)

order = payment.order

course = order.course

if not course.downloadable\_file:

raise Http404("Course file not found")

file\_path = os.path.join(settings.MEDIA\_ROOT, str(course.downloadable\_file))

if not os.path.exists(file\_path):

raise Http404("Course file not found")

# Set the appropriate filename using the original file name and extension

filename = os.path.basename(file\_path)

content\_type, encoding = mimetypes.guess\_type(filename)

response = FileResponse(open(file\_path, 'rb'), content\_type=content\_type)

response['Content-Disposition'] = f'attachment; filename="{filename}"'

# Print the response headers

for key, value in response.items():

print(f'{key}: {value}')

order.downloaded = True

order.save()

return response

except Payment.DoesNotExist:

raise Http404("Payment not found")

#paystack payment

# from rest\_framework.decorators import api\_view

# from rest\_framework.response import Response

# from rest\_framework import status

# from paystackapi.paystack import Paystack

# from paystackapi.transaction import Transaction

# @api\_view(['POST'])

# def payment\_list(request):

# try:

# order\_id = int(request.data.get('order'))

# except (TypeError, ValueError):

# return Response({'error': 'Invalid order ID'}, status=status.HTTP\_400\_BAD\_REQUEST)

# try:

# order = Order.objects.get(id=order\_id)

# except Order.DoesNotExist:

# return Response({'error': 'Order not found'}, status=status.HTTP\_404\_NOT\_FOUND)

# amount = int(order.price \* 100)

# order\_serializer = OrderSerializer(order)

# data={

# 'order': order\_serializer.data,

# 'amount': amount,

# 'card\_holder': request.data.get('card\_holder'),

# 'card\_number': request.data.get('card\_number'),

# 'card\_expiry\_date': request.data.get('card\_expiry\_date'),

# 'cvc': request.data.get('cvc'),

# }

# serializer = PaymentSerializer(data=data)

# if serializer.is\_valid():

# # Initialize the Paystack API with your secret key

# paystack = Paystack(secret\_key=settings.PAYSTACK\_SECRET\_KEY)

# # Create a transaction

# transaction = Transaction.initialize(

# amount=amount,

# # email=data['email'],

# # reference='PAYSTACK\_REFERENCE', # Provide a unique reference for the transaction

# card\_number=data['card\_number'],

# cvc=data['cvc'],

# expiry\_month=serializer.validated\_data['card\_expiry\_date'].month,

# expiry\_year=serializer.validated\_data['card\_expiry\_date'].year

# )

# if transaction.status:

# # Save the payment details in the Payment model

# payment = serializer.save(order=order)

# payment.transaction\_id = transaction.reference

# payment.save()

# return Response(serializer.data, status=status.HTTP\_201\_CREATED)

# else:

# return Response({'error': 'Payment failed'}, status=status.HTTP\_400\_BAD\_REQUEST)

# else:

# return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework import status

from django.shortcuts import get\_object\_or\_404

from .models import Order, Course

@api\_view(['GET'])

def check\_course\_download(request, course\_id):

try:

# Get the current user

user = request.user

# Retrieve the course ID from the request parameters

# course\_id = request.GET.get('course\_id')

print(course\_id)

# Retrieve the course object by its ID course = get\_object\_or\_404(Course, id=course\_id)

# Check if the user is authenticated

if not user.is\_authenticated:

return Response("User not authenticated", status=status.HTTP\_401\_UNAUTHORIZED)

# Check if the user has any orders for the given course that have been downloaded

order = Order.objects.filter(user=user, course=course, downloaded=True).first()

if not order:

return Response(False)

# User has downloaded the course

return Response(True)

except Http404 as e:

return Response(str(e), status=status.HTTP\_404\_NOT\_FOUND)

except Exception as e:

return Response(str(e), status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

class PaymentDetail(generics.RetrieveAPIView):

queryset = Payment.objects.all()

serializer\_class = PaymentSerializer

class ReviewList(generics.ListCreateAPIView):

queryset = Review.objects.all()

serializer\_class = ReviewSerializer

permission\_classes = [permissions.IsAuthenticated]

def perform\_create(self, serializer):

serializer.save(user=self.request.user)

from rest\_framework.decorators import api\_view, permission\_classes

from rest\_framework.response import Response

from rest\_framework import permissions, status

@api\_view(['POST'])

@permission\_classes([permissions.IsAuthenticated])

def create\_review(request):

course\_id = request.data.get('course\_id')

user = request.user

rating = request.data.get('rating')

comment = request.data.get('comment')

# Retrieve the course object from the database

try:

course = Course.objects.get(id=course\_id)

except Course.DoesNotExist:

return Response({'error': 'Course not found'}, status=status.HTTP\_404\_NOT\_FOUND)

# Create a new review object and save it to the database

review = Review.objects.create(course=course, user=user, rating=rating, content=comment)

review.save()

# Serialize the review object and return it as a JSON response

serializer = ReviewSerializer(review)

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

from rest\_framework.decorators import api\_view, permission\_classes

from rest\_framework.permissions import AllowAny

from rest\_framework.response import Response

from .models import Review, Course

@api\_view(['GET'])

@permission\_classes([AllowAny])

def course\_reviews(request, course\_id):

# Retrieve the course by its ID

try:

course = Course.objects.get(id=course\_id)

except Course.DoesNotExist:

return Response(status=404)

# Retrieve all the reviews for the course

reviews = Review.objects.filter(course=course)

# Serialize the reviews and return them in the response

serialized\_reviews = [{'user': review.user.username, 'content': review.content, 'rating': review.rating} for review in reviews]

return Response(serialized\_reviews)

class ReviewDetail(generics.RetrieveAPIView):

queryset = Review.objects.all()

serializer\_class = ReviewSerializer

from django.conf import settings

from rest\_framework.views import APIView

from rest\_framework.response import Response

class SubscriptionInfoView(APIView):

permission\_classes = [AllowAny]

def get(self, request):

subscription\_name = settings.SUBSCRIPTION\_NAME

subscription\_description = settings.SUBSCRIPTION\_DESCRIPTION

subscription\_price\_monthly = settings.SUBSCRIPTION\_PRICE\_MONTHLY

subscription\_price\_monthly\_discount = settings.SUBSCRIPTION\_PRICE\_MONTHLY\_DISCOUNT

subscription\_price\_yearly = settings.SUBSCRIPTION\_PRICE\_YEARLY

subscription\_price\_yearly\_discount = settings.SUBSCRIPTION\_PRICE\_YEARLY\_DISCOUNT

# Construct the response with the subscription information

response\_data = {

'name': subscription\_name,

'description': subscription\_description,

'price\_monthly': subscription\_price\_monthly,

'price\_monthly\_discount': subscription\_price\_monthly\_discount,

'price\_yearly': subscription\_price\_yearly,

'price\_yearly\_discount': subscription\_price\_yearly\_discount

}

return Response(response\_data)

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from .models import AllAccessSubscriber

from .serializers import AllAccessSubscriberSerializer

@api\_view(['GET'])

@permission\_classes([AllowAny])

def check\_all\_access\_subscription(request):

user\_id = user\_id = request.GET.get('userId')

print(user\_id)

active\_subscription = AllAccessSubscriber.objects.filter(user\_id=user\_id, is\_active=True).exists()

return Response({'has\_active\_subscription': active\_subscription})

from django.shortcuts import render, get\_object\_or\_404

from django.http import HttpResponse, JsonResponse

from django.conf import settings

import os

from .models import Course

def stream\_course(request, course\_id):

course = get\_object\_or\_404(Course, id=course\_id)

if course.downloadable\_file:

file\_path = course.downloadable\_file.path

filename = course.downloadable\_file.name

# Serve the file as a response

response = HttpResponse(content\_type="video/mp4")

response['Access-Control-Expose-Headers'] = 'Content-Disposition'

response['Content-Disposition'] = f'inline; filename="{filename}"' # Set the filename in the response header

with open(file\_path, 'rb') as file:

response.write(file.read())

# Construct the video URL using the media URL and file path

video\_url = request.build\_absolute\_uri(settings.MEDIA\_URL + filename)

# Include the video\_url and filename in the response data

data = {

'video\_url': video\_url,

'filename': filename

}

return JsonResponse(data)

else:

return render(request, 'error.html', {'message': 'No downloadable file available for this course.'})