**Django Developer Assignment**

**Vendor Management System with Performance Metrics**

1. Vendor Model

**class Vendor(models.Model):**

**name=models.CharField(max\_length=255)**

**contact\_details=models.TextField()**

**address=models.TextField()**

**vendor\_code=models.CharField(max\_length=50, unique=True)**

**on\_time\_delivery\_rate=models.FloatField(default=0)**

**quality\_rating=models.FloatField(default=0)**

**response\_time=models.FloatField(default=0)**

**fulfillment\_rate=models.FloatField(default=0)**

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

**return self.name**

2. Purchase Order (PO) Model

**class PurchaseOrder(models.Model):**

**vendor=models.ForeignKey('Vendor',on\_delete=models.CASCADE)**

**po\_number=models.CharField(max\_length=100,unique=True)**

**order\_date=models.DateTimeField()**

**delivery\_date=models.DateTimeField()**

**items=models.JSONField()**

**quantity=models.IntegerField()**

**status=models.CharField(max\_length=50)**

**quality\_rating=models.FloatField(null=True)**

**issue\_date=models.DateTimeField()**

**acknowledgment\_date=models.DateTimeField(null=True)**

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

**return self.po\_number**

3. Historical Performance Model

**class PerformanceRecord(models.Model):**

**vendor=models.ForeignKey('Vendor',on\_delete=models.CASCADE)**

**date=models.DateTimeField()**

**on\_time\_delivery\_rate=models.FloatField()**

**quality\_rating\_avg=models.FloatField()**

**average\_response\_time=models.FloatField()**

**fulfillment\_rate=models.FloatField()**

**class Meta:**

**verbose\_name=\_('Performance Record')**

**verbose\_name\_plural=\_('Performance Records')**

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

**return f"Performance Record for {self.vendor.name} on {self.date.strftime('%Y-%m-%d')}”**

Admin.py

from app.models import \*

admin.site.register(Vendor)

admin.site.register(PurchaseOrder)

admin.site.register(PerformanceRecord)

After completing the models.py and Registration in Admin.py do that Migration and createsuperuser

* **python manage.py migrate**
* **python manage.py makemigrations**
* **python manage.py migrate**
* **python manage.py createsuperuser**

**After that Create Serializers.py inside the app folder**

from rest\_framework import serializers

from app.models import \*

from app.serializers import \*

class VendorSerializer(serializers.ModelSerializer):

    class Meta:

model=Vendor

        fields='\_\_all\_\_'

*# fields = ['id','name','contact\_details','address','vendor\_code']*

**1. class VendorSerializer(serializers.ModelSerializer):**

This line defines a serializer class named VendorSerializer. It inherits from serializers.ModelSerializer, which is provided by Django REST Framework and makes it easy to serialize and deserialize Django model instances.

**class Meta:**

**model = Vendor**

**fields = '\_\_all\_\_'**

Within the Meta inner class, you specify metadata for the serializer. Here:

* model = Vendor: This specifies the Django model (Vendor) that the serializer should be based on.
* fields = '\_\_all\_\_': This indicates that all fields of the Vendor model should be included in the serialization process. Using '\_\_all\_\_' is a shortcut that automatically includes all fields of the model.

**# fields = ['id', 'name', 'contact\_details', 'address', 'vendor\_code']**

In this case, only the specified fields (id, name, contact\_details, address, vendor\_code) would be included in the serialization process.

**2. class PurchaseOrderSerializer(serializers.ModelSerializer):**

Similar to the previous serializer, this line defines a serializer class named PurchaseOrderSerializer that inherits from serializers.ModelSerializer.

**class Meta:**

**model = PurchaseOrder**

**fields = '\_\_all\_\_'**

Within the Meta inner class:

* model = PurchaseOrder: This specifies the Django model (PurchaseOrder) that the serializer should be based on.
* fields = '\_\_all\_\_': This indicates that all fields of the PurchaseOrder model should be included in the serialization process. Using '\_\_all\_\_' is a shortcut that automatically includes all fields of the model.

**# fields = ['id', 'po\_number', 'vendor', 'order\_date', 'items', 'quantity', 'status']**

In this case, only the specified fields (**id**, **po\_number**, **vendor**, **order\_date**, **items**, **quantity**, **status**) would be included in the serialization process.

**3. class VendorPerformanceSerializer(serializers.ModelSerializer):**

This line declares a serializer class named VendorPerformanceSerializer, extending serializers.ModelSerializer from Django REST Framework.

**class Meta:**

**model = Vendor**

**fields = '\_\_all\_\_'**

Within the Meta inner class:

* model = Vendor: This specifies the Django model that the serializer is based on, which in this case is the Vendor model.
* fields = '\_\_all\_\_': This line indicates that all fields of the Vendor model should be included in the serialization process. '\_\_all\_\_' is a shortcut to include all fields.

**# fields = ['id', 'on\_time\_delivery\_rate', 'quality\_rating', 'response\_time', 'fulfilment\_rate']**

If you choose this option, only the specified fields (id, on\_time\_delivery\_rate, quality\_rating, response\_time, fulfilment\_rate) will be included in the serialization process**.**

**URLS.PY AND VIEWS.PY** :

First import the models and serializers in views.py

from app.models import \*

from app.serializers import \*

from rest\_framework.decorators import APIView

class VendorListCreateAPIView(APIView):

The code class VendorListCreateAPIView(APIView): is defining a new Django REST framework API view class named VendorListCreateAThis is a method definition for the get request in the VendorListCreateAPIView class. Here's a breakdown of what it does:

path('api/vendors/', VendorListCreateAPIView.as\_view()),

This line of code defines a URL pattern in Django using the path function from the django.urls module.

Here's a breakdown of what it does:

1. path('api/vendors/', ...): Defines a URL pattern that matches the string 'api/vendors/'.

2. VendorListCreateAPIView.as\_view(): Specifies the view function that will handle requests to this URL. In this case, it's the VendorListCreateAPIView class, which is a Django REST framework view.

The as\_view() method is a class method that returns a view function that can be used in URL patterns.

 def get(self,request):

        vendors=Vendor.objects.all()

        serializer=VendorSerializer(vendors,many=True)

        return Response(serializer.data)

1. def get(self, request):: Defines a method named get that takes two parameters: self (a reference to the instance of the class) and request (the incoming HTTP request).

2. vendors = Vendor.objects.all(): Retrieves all instances of the Vendor model from the database using the Django ORM (Object-Relational Mapping) system.

3. serializer = VendorSerializer(vendors, many=True): Creates an instance of the VendorSerializer class, passing the vendors queryset and the many=True parameter. The many=True parameter indicates that the serializer is handling a collection of objects (in this case, multiple vendors).

4. return Response(serializer.data): Returns an HTTP response containing the serialized data. The serializer.data attribute contains the serialized data in a format suitable for transmission (e.g., JSON).PIView that inherits from APIView.

def post(self, request):

        serializer=VendorSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response({'insert':'Data inserted successfull'})

        else:

            return Response({'Error':'Data insertion Error'})

This is a method definition for the post request in the VendorListCreateAPIView class. Here's a breakdown of what it does:

1. def post(self, request):: Defines a method named post that takes two parameters: self (a reference to the instance of the class) and request (the incoming HTTP request).

2. serializer = VendorSerializer(data=request.data): Creates an instance of the VendorSerializer class, passing the data from the incoming request (request.data) to the serializer.

3. if serializer.is\_valid():: Calls the is\_valid method on the serializer to validate the data. If the data is valid, the code within the if block is executed.

4. serializer.save(): Saves the validated data to the database using the serializer's save method.

5. return Response({'insert':'Data inserted successfully'}): Returns an HTTP response with a JSON payload indicating that the data was inserted successfully.

6. else: return Response({'Error':'Data insertion Error'}): If the data is not valid (i.e., the is\_valid method returns False), returns an HTTP response with a JSON payload indicating that there was an error inserting the data.

class VendorRetrieveUpdateDestroyAPIView(APIView):

- class VendorRetrieveUpdateDestroyAPIView(APIView): Defines a new class named VendorRetrieveUpdateDestroyAPIView that inherits from APIView.

- APIView is a base class provided by Django REST framework for building API views.

path('api/vendors/<int:vendor\_id>/', VendorRetrieveUpdateDestroyAPIView.as\_view()),

This URL pattern would map GET, PUT, PATCH, and DELETE requests to the get(), put(), patch(), and delete() methods of the VendorRetrieveUpdateDestroyAPIView class, respectively.

def get(self,request,vendor\_id):

        vendor=Vendor.objects.get(pk=vendor\_id)

        serializer=VendorSerializer(vendor)

        return Response(serializer.data)

This is a method definition for the get request in the VendorRetrieveUpdateDestroyAPIView class. Here's a breakdown of what it does:

1. def get(self, request, vendor\_id):: Defines a method named get that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- vendor\_id: the ID of the vendor to retrieve (passed from the URL pattern)

2. vendor = Vendor.objects.get(pk=vendor\_id): Retrieves the Vendor instance with the primary key (pk) equal to vendor\_id from the database using the Django ORM.

3. serializer = VendorSerializer(vendor): Creates an instance of the VendorSerializer class, passing the retrieved vendor instance to the serializer.

4. return Response(serializer.data): Returns an HTTP response containing the serialized data. The serializer.data attribute contains the serialized data in a format suitable for transmission (e.g., JSON).

def put(self,request,vendor\_id):

        vendor=Vendor.objects.get(pk=vendor\_id)

        serializer=VendorSerializer(vendor, data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response({'insert':'Data inserted successfull'})

        else:

            return Response({'Error':'Data insertion Error'})

This is a method definition for the put request in the VendorRetrieveUpdateDestroyAPIView class. Here's a breakdown of what it does:

1. def put(self, request, vendor\_id):: Defines a method named put that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- vendor\_id: the ID of the vendor to update (passed from the URL pattern)

2. vendor = Vendor.objects.get(pk=vendor\_id): Retrieves the Vendor instance with the primary key (pk) equal to vendor\_id from the database using the Django ORM.

3. serializer = VendorSerializer(vendor, data=request.data): Creates an instance of the VendorSerializer class, passing the retrieved vendor instance and the data from the incoming request (request.data) to the serializer.

4. if serializer.is\_valid():: Calls the is\_valid method on the serializer to validate the data. If the data is valid, the code within the if block is executed.

5. serializer.save(): Saves the validated data to the database using the serializer's save method, updating the existing vendor instance.

6. return Response({'insert':'Data inserted successfully'}): Returns an HTTP response with a JSON payload indicating that the data was updated successfully.

7. else: return Response({'Error':'Data insertion Error'}): If the data is not valid (i.e., the is\_valid method returns False), returns an HTTP response with a JSON payload indicating that there was an error updating the data.

 def delete(self,request,vendor\_id):

        vendor=Vendor.objects.get(pk=vendor\_id)

        vendor.delete()

        return Response({'deletion':'Data is deleted'})

That's correct! This method definition handles the DELETE request for the VendorRetrieveUpdateDestroyAPIView class. It takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- vendor\_id: the ID of the vendor to delete (passed from the URL pattern)

The method performs the following actions:

1. Retrieves the Vendor instance with the primary key (pk) equal to vendor\_id from the database using the Django ORM.

2. Deletes the retrieved Vendor instance from the database.

3. Returns an HTTP response with a JSON payload indicating that the data was deleted successfully.

class PurchaseOrderListCreateAPIView(APIView):

This line of code defines a new Django REST framework API view class named PurchaseOrderListCreateAPIView that inherits from APIView.

This view is designed to handle CRUD (Create, Retrieve, Update, Delete) operations for purchase orders. The name of the class suggests that it will handle:

- List (GET): Retrieve a list of purchase orders

- Create (POST): Create a new purchase order

 path('api/purchase\_orders/', PurchaseOrderListCreateAPIView.as\_view()),

This URL pattern would map GET and POST requests to the get() and post() methods of the PurchaseOrderListCreateAPIView class, respectively.

def get(self,request):

        vendor\_id=request.query\_params.get('vendor\_id')

        if vendor\_id:

            purchase\_orders=PurchaseOrder.objects.filter(vendor\_reference=vendor\_id)

        else:

            purchase\_orders=PurchaseOrder.objects.all()

        serializer=PurchaseOrderSerializer(purchase\_orders,many=True)

        return Response({'get':'Data display successfull'})

This is a method definition for the get request in the PurchaseOrderListCreateAPIView class. Here's a breakdown of what it does:

1. def get(self, request):: Defines a method named get that takes two parameters: self (a reference to the instance of the class) and request (the incoming HTTP request).

2. vendor\_id = request.query\_params.get('vendor\_id'): Retrieves the vendor\_id parameter from the query parameters in the incoming request. If the parameter is present, its value is stored in the vendor\_id variable.

3. if vendor\_id: ... else: ...: Checks if a vendor\_id was provided. If it was, the code inside the if block is executed. If not, the code inside the else block is executed.

4. purchase\_orders = PurchaseOrder.objects.filter(vendor\_reference=vendor\_id): If a vendor\_id was provided, retrieves all PurchaseOrder instances from the database where the vendor\_reference field matches the provided vendor\_id.

5. purchase\_orders = PurchaseOrder.objects.all(): If no vendor\_id was provided, retrieves all PurchaseOrder instances from the database.

6. serializer = PurchaseOrderSerializer(purchase\_orders, many=True): Creates an instance of the PurchaseOrderSerializer class, passing the retrieved purchase\_orders and the many=True parameter to the serializer.

7. return Response({'get': 'Data display successful'}): Returns an HTTP response with a JSON payload indicating that the data was retrieved successfully.

 def post(self,request):

        serializer=PurchaseOrderSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response({'insert':'Data inserted successfull'})

        else:

            return Response({'Error':'Data insertion Error'})

This is a method definition for the post request in the PurchaseOrderListCreateAPIView class. Here's a breakdown of what it does:

1. def post(self, request):: Defines a method named post that takes two parameters: self (a reference to the instance of the class) and request (the incoming HTTP request).

2. serializer = PurchaseOrderSerializer(data=request.data): Creates an instance of the PurchaseOrderSerializer class, passing the data from the incoming request (request.data) to the serializer.

3. if serializer.is\_valid():: Calls the is\_valid method on the serializer to validate the data. If the data is valid, the code within the if block is executed.

4. serializer.save(): Saves the validated data to the database using the serializer's save method, creating a new PurchaseOrder instance.

5. return Response({'insert': 'Data inserted successfully'}): Returns an HTTP response with a JSON payload indicating that the data was inserted successfully.

6. else: return Response({'Error': 'Data insertion Error'}): If the data is not valid (i.e., the is\_valid method returns False), returns an HTTP response with a JSON payload indicating that there was an error inserting the data.

class PurchaseOrderRetrieveUpdateDestroyAPIView(APIView):

This view is designed to handle CRUD (Create, Retrieve, Update, Delete) operations for a single purchase order. The name of the class suggests that it will handle:

- Retrieve (GET): Retrieve a single purchase order instance

- Update (PUT/PATCH): Update a single purchase order instance

- Destroy (DELETE): Delete a single purchase order instance

path('api/purchase\_orders/<int:po\_id>/', PurchaseOrderRetrieveUpdateDestroyAPIView.as\_view()),

This URL pattern would map GET, PUT, PATCH, and DELETE requests to the get(), put(), patch(), and delete() methods of the PurchaseOrderRetrieveUpdateDestroyAPIView class, respectively.

 def get(self,request,po\_id):

        purchase\_order=PurchaseOrder.objects.get(pk=po\_id)

        serializer=PurchaseOrderSerializer(purchase\_order)

        return Response(serializer.data)

This is a method definition for the get request in the PurchaseOrderRetrieveUpdateDestroyAPIView class. Here's a breakdown of what it does:

1. def get(self, request, po\_id):: Defines a method named get that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- po\_id: the ID of the purchase order to retrieve (passed from the URL pattern)

2. purchase\_order = PurchaseOrder.objects.get(pk=po\_id): Retrieves the PurchaseOrder instance with the primary key (pk) equal to po\_id from the database using the Django ORM.

3. serializer = PurchaseOrderSerializer(purchase\_order): Creates an instance of the PurchaseOrderSerializer class, passing the retrieved purchase\_order instance to the serializer.

4. return Response(serializer.data): Returns an HTTP response with the serialized data.

def put(self,request,po\_id):

        purchase\_order=PurchaseOrder.objects.get(pk=po\_id)

        serializer=PurchaseOrderSerializer(purchase\_order,data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response({'insert':'Data inserted successfull'})

        else:

            return Response({'Error':'Data insertion Error'})

This is a method definition for the put request in the PurchaseOrderRetrieveUpdateDestroyAPIView class. Here's a breakdown of what it does:

1. def put(self, request, po\_id):: Defines a method named put that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- po\_id: the ID of the purchase order to update (passed from the URL pattern)

2. purchase\_order = PurchaseOrder.objects.get(pk=po\_id): Retrieves the PurchaseOrder instance with the primary key (pk) equal to po\_id from the database using the Django ORM.

3. serializer = PurchaseOrderSerializer(purchase\_order, data=request.data): Creates an instance of the PurchaseOrderSerializer class, passing the retrieved purchase\_order instance and the data from the incoming request (request.data) to the serializer.

4. if serializer.is\_valid():: Calls the is\_valid method on the serializer to validate the data. If the data is valid, the code within the if block is executed.

5. serializer.save(): Saves the validated data to the database using the serializer's save method, updating the existing PurchaseOrder instance.

6. return Response({'insert': 'Data inserted successfully'}): Returns an HTTP response with a JSON payload indicating that the data was updated successfully.

7. else: return Response({'Error': 'Data insertion Error'}): If the data is not valid (i.e., the is\_valid method returns False), returns an HTTP response with a JSON payload indicating that there was an error updating the data.

def delete(self,request,po\_id):

        purchase\_order=PurchaseOrder.objects.get(pk=po\_id)

        purchase\_order.delete()

        return Response({'deletion':'Data is deleted'})

This is a method definition for the delete request in the PurchaseOrderRetrieveUpdateDestroyAPIView class. Here's a breakdown of what it does:

1. def delete(self, request, po\_id):: Defines a method named delete that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- po\_id: the ID of the purchase order to delete (passed from the URL pattern)

2. purchase\_order = PurchaseOrder.objects.get(pk=po\_id): Retrieves the PurchaseOrder instance with the primary key (pk) equal to po\_id from the database using the Django ORM.

3. purchase\_order.delete(): Deletes the retrieved PurchaseOrder instance from the database.

4. return Response({'deletion': 'Data is deleted'}): Returns an HTTP response with a JSON payload indicating that the data was deleted successfully.

class VendorPerformanceAPIView(APIView):

Here's a breakdown of the class:

- class VendorPerformanceAPIView(APIView): Defines a new class named VendorPerformanceAPIView that inherits from APIView.

- APIView is a base class provided by Django REST framework for building API views.

 path('api/vendors/<int:vendor\_id>/performance/',VendorPerformanceAPIView.as\_view()),

his URL pattern would map requests to the VendorPerformanceAPIView class, which would then handle the request and return a response.

def get(self,request,vendor\_id):

        try:

            vendor=Vendor.objects.get(pk=vendor\_id)

            serializer=VendorPerformanceSerializer(vendor)

            return Response(serializer.data)

        except Vendor.DoesNotExist:

            return Response({"error":"Vendor not found"})

This is a Django REST framework view function that handles a GET request to retrieve vendor performance data. Here's a breakdown of what it does:

1. def get(self, request, vendor\_id):: This defines a function named get that takes three parameters:

- self: a reference to the instance of the class

- request: the incoming HTTP request

- vendor\_id: the ID of the vendor to retrieve performance data for

2. try:: This starts a try-except block to handle potential errors.

3. vendor = Vendor.objects.get(pk=vendor\_id): This retrieves the Vendor instance with the primary key (pk) equal to vendor\_id from the database using the Django ORM.

4. serializer = VendorPerformanceSerializer(vendor): This creates an instance of the VendorPerformanceSerializer class, passing the retrieved vendor instance to the serializer.

5. return Response(serializer.data): This returns an HTTP response with the serialized data.

6. except Vendor.DoesNotExist:: This catches the Vendor.DoesNotExist exception, which is raised if the vendor with the specified vendor\_id is not found in the database.

7. return Response({"error": "Vendor not found"}): This returns an HTTP response with a JSON payload indicating that the vendor was not found.

*#Back End Logic Here........*

class CalculateOnTimeDeliveryRate(APIView):

Here's a breakdown of the class:

- class CalculateOnTimeDeliveryRate(APIView): Defines a new class named CalculateOnTimeDeliveryRate that inherits from APIView.

path('api/calculate\_on\_time\_delivery\_rate/',CalculateOnTimeDeliveryRate.as\_view(),

This URL pattern would map requests to the CalculateOnTimeDeliveryRate class, which would then handle the request and return a response.

def post(self,request):

*# Retrieve the purchase order ID from the request data*

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

        try:

*# Retrieve the purchase order*

            po=PurchaseOrder.objects.get(id=po\_id)

            if po.status=='completed':

*# Retrieve the vendor*

                vendor=po.vendor

*# Count the number of completed POs delivered on or before delivery\_date*

                completed\_orders=PurchaseOrder.objects.filter(vendor=vendor,status='completed')

                on\_time\_orders=completed\_orders.filter(delivery\_date\_\_lte=po.delivery\_date)

                on\_time\_delivery\_rate=(on\_time\_orders.count()/completed\_orders.count())\*100

*# Update vendor's On-Time Delivery Rate*

                vendor.on\_time\_delivery\_rate=on\_time\_delivery\_rate

                vendor.save()

                return Response({'message':'On-Time Delivery Rate calculated and updated successfully.'})

            else:

                return Response({'error':'The purchase order status is not completed.'})

        except PurchaseOrder.DoesNotExist:

            return Response({'error':'Purchase order not found.'})

This is a Django REST framework view function that handles a POST request to calculate and update the on-time delivery rate for a vendor. Here's a breakdown of what it does:

1. def post(self, request):: This defines a function named post that takes two parameters: self (a reference to the instance of the class) and request (the incoming HTTP request).

2. po\_id = request.data.get('po\_id'): This retrieves the purchase order ID from the request data.

3. try:: This starts a try-except block to handle potential errors.

4. po = PurchaseOrder.objects.get(id=po\_id): This retrieves the PurchaseOrder instance with the ID equal to po\_id from the database.

5. if po.status == 'completed':: This checks if the purchase order status is 'completed'. If true, it proceeds with the calculation.

6. vendor = po.vendor: This retrieves the vendor associated with the purchase order.

7. completed\_orders = PurchaseOrder.objects.filter(vendor=vendor, status='completed'): This retrieves all completed purchase orders for the vendor.

8. on\_time\_orders = completed\_orders.filter(delivery\_date\_\_lte=po.delivery\_date): This filters the completed orders to only include those delivered on or before the delivery date.

9. on\_time\_delivery\_rate = (on\_time\_orders.count() / completed\_orders.count()) \* 100: This calculates the on-time delivery rate as a percentage.

10. vendor.on\_time\_delivery\_rate = on\_time\_delivery\_rate: This updates the vendor's on-time delivery rate.

11. vendor.save(): This saves the updated vendor instance to the database.

12. return Response({'message': 'On-Time Delivery Rate calculated and updated successfully.'}): This returns a successful response message.

13. else: return Response({'error': 'The purchase order status is not completed.'}): If the purchase order status is not 'completed', it returns an error response.

14. except PurchaseOrder.DoesNotExist: return Response({'error': 'Purchase order not found.'}): If the purchase order with the specified ID is not found, it returns an error response.

class UpdateQualityRatingAverage(APIView):

Here's a breakdown of the class:

- class UpdateAverageResponseTime(APIView): Defines a new class named UpdateAverageResponseTime that inherits from APIView.

path('api/update\_quality\_rating\_average/', UpdateQualityRatingAverage.as\_view()),

This URL pattern would map requests to the UpdateAverageResponseTime class, which would then handle the request and return a response.

def post(self,request):

*# Retrieve the purchase order ID and quality rating from the request data*

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

        quality\_rating=request.data.get('quality\_rating')

        try:

*# Retrieve the purchase order*

            po=PurchaseOrder.objects.get(id=po\_id)

*# Ensure the purchase order is completed and quality rating is provided*

            if po.status=='completed' and quality\_rating is not None:

*# Retrieve the vendor*

                vendor=po.vendor

*# Calculate the average of all quality\_rating values for completed POs of the vendor*

                completed\_orders=PurchaseOrder.objects.filter(vendor=vendor,status='completed').exclude(quality\_rating\_\_isnull=True)

                total\_ratings=completed\_orders.aggregate(total\_rating=models.Sum('quality\_rating'))['total\_rating']

                num\_completed\_orders=completed\_orders.count()

                quality\_rating\_average=total\_ratings/num\_completed\_orders if num\_completed\_orders > 0 else 0

*# Update vendor's Quality Rating Average*

                vendor.quality\_rating=quality\_rating\_average

                vendor.save()

                return Response({'message':'Quality Rating Average updated successfully.'})

            else:

                return Response({'error':'The purchase order status is not completed or quality rating is not provided.'})

        except PurchaseOrder.DoesNotExist:

            return Response({'error':'Purchase order not found.'})

This is a Django REST framework view function that handles a POST request to update the quality rating average for a vendor. Here's a breakdown of what it does:

1. po\_id = request.data.get('po\_id'): Retrieves the purchase order ID from the request data.

2. quality\_rating = request.data.get('quality\_rating'): Retrieves the quality rating from the request data.

3. try:: Starts a try-except block to handle potential errors.

4. po = PurchaseOrder.objects.get(id=po\_id): Retrieves the PurchaseOrder instance with the ID equal to po\_id from the database.

5. if po.status == 'completed' and quality\_rating is not None:: Checks if the purchase order status is 'completed' and a quality rating is provided. If true, it proceeds with the calculation.

6. vendor = po.vendor: Retrieves the vendor associated with the purchase order.

7. completed\_orders = PurchaseOrder.objects.filter(vendor=vendor, status='completed').exclude(quality\_rating\_\_isnull=True): Retrieves all completed purchase orders for the vendor with a quality rating.

8. total\_ratings = completed\_orders.aggregate(total\_rating=models.Sum('quality\_rating'))['total\_rating']: Calculates the total of all quality ratings for the completed orders.

9. num\_completed\_orders = completed\_orders.count(): Counts the number of completed orders.

10. quality\_rating\_average = total\_ratings / num\_completed\_orders if num\_completed\_orders > 0 else 0: Calculates the average quality rating.

11. vendor.quality\_rating = quality\_rating\_average: Updates the vendor's quality rating average.

12. vendor.save(): Saves the updated vendor instance to the database.

13. return Response({'message': 'Quality Rating Average updated successfully.'}): Returns a successful response message.

14. else: return Response({'error': 'The purchase order status is not completed or quality rating is not provided.'}): If the purchase order status is not 'completed' or a quality rating is not provided, it returns an error response.

15. except PurchaseOrder.DoesNotExist: return Response({'error': 'Purchase order not found.'}): If the purchase order with the specified ID is not found, it returns an error response.

class UpdateAverageResponseTime(APIView):

Here's a breakdown of the class:

- class UpdateAverageResponseTime(APIView): Defines a new class named UpdateAverageResponseTime that inherits from APIView.

path('api/update\_average\_response\_time/', UpdateAverageResponseTime.as\_view()),

This URL pattern would map requests to the UpdateAverageResponseTime class, which would then handle the request and return a response.

def post(self,request):

*# Retrieve the purchase order ID and acknowledgment date from the request data*

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

        acknowledgment\_date\_str=request.data.get('acknowledgment\_date')

        try:

*# Retrieve the purchase order*

            po=PurchaseOrder.objects.get(id=po\_id)

*# Ensure acknowledgment date is provided*

            if acknowledgment\_date\_str:

                acknowledgment\_date=datetime.strptime(acknowledgment\_date\_str,'%Y-%m-%d %H:%M:%S')

*# Retrieve the vendor*

                vendor=po.vendor

*# Compute the time difference between issue\_date and acknowledgment\_date for each PO*

                time\_diff=acknowledgment\_date-po.issue\_date

*# Update vendor's Average Response Time*

                total\_response\_time=vendor.response\_time\*vendor.num\_acknowledgments

                total\_response\_time+=time\_diff.total\_seconds()

                vendor.num\_acknowledgments+=1

                vendor.response\_time=total\_response\_time/vendor.num\_acknowledgments

                vendor.save()

                return Response({'message':'Average Response Time updated successfully.'})

            else:

                return Response({'error':'Acknowledgment date is required.'})

        except PurchaseOrder.DoesNotExist:

            return Response({'error':'Purchase order not found.'})

This is a Django REST framework view function that handles a POST request to update the average response time for a vendor. Here's a breakdown of what it does:

1. po\_id = request.data.get('po\_id'): Retrieves the purchase order ID from the request data.

2. acknowledgment\_date\_str = request.data.get('acknowledgment\_date'): Retrieves the acknowledgment date from the request data as a string.

3. try:: Starts a try-except block to handle potential errors.

4. po = PurchaseOrder.objects.get(id=po\_id): Retrieves the PurchaseOrder instance with the ID equal to po\_id from the database.

5. if acknowledgment\_date\_str:: Checks if an acknowledgment date is provided. If true, it proceeds with the calculation.

6. acknowledgment\_date = datetime.strptime(acknowledgment\_date\_str, '%Y-%m-%d %H:%M:%S'): Converts the acknowledgment date string to a datetime object.

7. vendor = po.vendor: Retrieves the vendor associated with the purchase order.

8. time\_diff = acknowledgment\_date - po.issue\_date: Computes the time difference between the issue date and acknowledgment date.

9. total\_response\_time = vendor.response\_time \* vendor.num\_acknowledgments: Calculates the total response time.

10. total\_response\_time += time\_diff.total\_seconds(): Adds the time difference to the total response time.

11. vendor.num\_acknowledgments += 1: Increments the number of acknowledgments.

12. vendor.response\_time = total\_response\_time / vendor.num\_acknowledgments: Updates the average response time.

13. vendor.save(): Saves the updated vendor instance to the database.

14. return Response({'message': 'Average Response Time updated successfully.'}): Returns a successful response message.

15. else: return Response({'error': 'Acknowledgment date is required.'}): If an acknowledgment date is not provided, it returns an error response.

16. except PurchaseOrder.DoesNotExist: return Response({'error': 'Purchase order not found.'}): If the purchase order with the specified ID is not found, it returns an error response.

class UpdateFulfilmentRate(APIView):

Here's a breakdown of the class:

- class UpdateFulfilmentRate(APIView): Defines a new class named UpdateFulfilmentRate that inherits from APIView.

path('api/update\_fulfilment\_rate/', UpdateFulfilmentRate.as\_view()),

This URL pattern would map requests to the UpdateFulfilmentRate class, which would then handle the request and return a response.

class UpdateFulfilmentRate(APIView):

    def post(self,request):

*# Retrieve the purchase order ID and new status from the request data*

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

        new\_status=request.data.get('new\_status')

        try:

*# Retrieve the purchase order*

            po=PurchaseOrder.objects.get(id=po\_id)

*# Retrieve the vendor*

            vendor=po.vendor

*# Update the fulfilment rate upon any change in PO status*

            if new\_status=='completed':

*# Check if the PO was successfully fulfilled*

                if po.issues==0:

                    vendor.successful\_orders+=1

*# Increment the total number of orders issued to the vendor*

                vendor.total\_orders+=1

*# Calculate the fulfilment rate*

            fulfilment\_rate=(vendor.successful\_orders/vendor.total\_orders)\*100 if vendor.total\_orders > 0 else 0

*# Update vendor's Fulfilment Rate*

            vendor.fulfilment\_rate=fulfilment\_rate

            vendor.save()

            return Response({'message':'Fulfilment Rate updated successfully.'})

        except PurchaseOrder.DoesNotExist:

            return Response({'error':'Purchase order not found.'})

This is a Django REST framework view function that handles a POST request to update the fulfillment rate for a vendor. Here's a breakdown of what it does:

1. po\_id = request.data.get('po\_id'): Retrieves the purchase order ID from the request data.

2. new\_status = request.data.get('new\_status'): Retrieves the new status from the request data.

3. try:: Starts a try-except block to handle potential errors.

4. po = PurchaseOrder.objects.get(id=po\_id): Retrieves the PurchaseOrder instance with the ID equal to po\_id from the database.

5. vendor = po.vendor: Retrieves the vendor associated with the purchase order.

6. if new\_status == 'completed':: Checks if the new status is 'completed'. If true, it proceeds with the calculation.

7. if po.issues == 0:: Checks if there are no issues with the purchase order. If true, it increments the successful orders count.

8. vendor.successful\_orders += 1: Increments the successful orders count for the vendor.

9. vendor.total\_orders += 1: Increments the total orders count for the vendor.

10. fulfilment\_rate = (vendor.successful\_orders / vendor.total\_orders) \* 100 if vendor.total\_orders > 0 else 0: Calculates the fulfillment rate.

11. vendor.fulfilment\_rate = fulfilment\_rate: Updates the vendor's fulfillment rate.

12. vendor.save(): Saves the updated vendor instance to the database.

13. return Response({'message': 'Fulfilment Rate updated successfully.'}): Returns a successful response message.

14. except PurchaseOrder.DoesNotExist: return Response({'error': 'Purchase order not found.'}): If the purchase order with the specified ID is not found, it returns an error response.

*# API Endpoint Implementation here....*

class VendorPerformanceEndpoint(APIView):

Here's a breakdown of the class:

- class VendorPerformanceEndpoint(APIView): Defines a new class named VendorPerformanceEndpoint that inherits from APIView.

path('api/vendors/<int:vendor\_id>/performance/',VendorPerformanceEndpoint.as\_view

)),

This URL pattern would map requests to the VendorPerformanceEndpoint class, which would then handle the request and return a response.

   def get(self,request,vendor\_id):

        try:

*# Retrieve the vendor*

            vendor=Vendor.objects.get(pk=vendor\_id)

*# Serialize vendor's performance metrics*

            serializer=VendorPerformanceSerializer(vendor)

            return Response({'insert':'Data inserted successfull'})

        except Vendor.DoesNotExist:

            return Response({'error':'Vendor not found.'})

This is a Django REST framework view function that handles a GET request to retrieve a vendor's performance metrics. Here's a breakdown of what it does:

1. try:: Starts a try-except block to handle potential errors.

2. vendor = Vendor.objects.get(pk=vendor\_id): Retrieves the Vendor instance with the primary key equal to vendor\_id from the database.

3. serializer = VendorPerformanceSerializer(vendor): Serializes the vendor's performance metrics using the VendorPerformanceSerializer class.

4. return Response({'insert':'Data inserted successfully'}): Returns a successful response message. However, the message indicates that data was inserted, which is not the case in a GET request. It should be changed to something like 'Vendor performance metrics retrieved successfully'.

5. except Vendor.DoesNotExist: return Response({'error':'Vendor not found.'}): If the vendor with the specified ID is not found, it returns an error response.

Note: The get method should actually return the serialized vendor performance metrics, not just a success message. The correct implementation should be:

def get(self, request, vendor\_id):

try:

vendor = Vendor.objects.get(pk=vendor\_id)

serializer = VendorPerformanceSerializer(vendor)

return Response(serializer.data)

except Vendor.DoesNotExist:

return Response({'error': 'Vendor not found.'})

This will return the serialized vendor performance metrics in the response body.

class UpdateAcknowledgmentEndpoint(APIView):

Here's a breakdown of the class:

- class UpdateAcknowledgmentEndpoint(APIView): Defines a new class named UpdateAcknowledgmentEndpoint that inherits from APIView.

path('api/purchase\_orders/<int:po\_id>/acknowledge/',UpdateAcknowledgmentEndpoint.as\_view()),

This URL pattern would map requests to the UpdateAcknowledgmentEndpoint class, which would then handle the request and return a response.

The methods defined in this class would depend on the specific requirements of the API, but could include:

- post(self, request): Handle POST requests to update the acknowledgment status.

- put(self, request): Handle PUT requests to update the acknowledgment status with additional data provided in the request body.

 def post(self,request,po\_id):

        try:

*# Retrieve the purchase order*

            po=PurchaseOrder.objects.get(pk=po\_id)

*# Update acknowledgment date*

            acknowledgment\_date\_str=request.data.get('acknowledgment\_date')

            if acknowledgment\_date\_str:

                acknowledgment\_date=datetime.strptime(acknowledgment\_date\_str,'%Y-%m-%d')

                po.acknowledgment\_date=acknowledgment\_date

                po.save()

*# Trigger recalculation of average\_response\_time*

                po.vendor.calculate\_average\_response\_time()

                return Response({'message':'Acknowledgment updated successfully.'})

            else:

                return Response({'error':'Acknowledgment date is required.'})

        except PurchaseOrder.DoesNotExist:

            return Response({'error':'Purchase order not found.'})

This is a Django REST framework view function that handles a POST request to update the acknowledgment date of a purchase order. Here's a breakdown of what it does:

1. try:: Starts a try-except block to handle potential errors.

2. po = PurchaseOrder.objects.get(pk=po\_id): Retrieves the PurchaseOrder instance with the primary key equal to po\_id from the database.

3. acknowledgment\_date\_str = request.data.get('acknowledgment\_date'): Retrieves the acknowledgment date from the request data.

4. if acknowledgment\_date\_str:: Checks if an acknowledgment date is provided. If true, it proceeds with the update.

5. acknowledgment\_date = datetime.strptime(acknowledgment\_date\_str, '%Y-%m-%d'): Converts the acknowledgment date string to a datetime object.

6. po.acknowledgment\_date = acknowledgment\_date: Updates the acknowledgment date of the purchase order.

7. po.save(): Saves the updated purchase order instance to the database.

8. po.vendor.calculate\_average\_response\_time(): Triggers the recalculation of the average response time for the vendor associated with the purchase order.

9. return Response({'message': 'Acknowledgment updated successfully.'}): Returns a successful response message.

10. else: return Response({'error': 'Acknowledgment date is required.'}): If an acknowledgment date is not provided, it returns an error response.

11. except PurchaseOrder.DoesNotExist: return Response({'error': 'Purchase order not found.'}): If the purchase order with the specified ID is not found, it returns an error response.

*# Additional Technical Considerations*

*# Efficient Calculation:*

class UpdateFulfilmentRate(APIView):

This view is likely designed to handle requests related to updating the fulfillment rate of a vendor or a purchase order.

Here's a breakdown of the class:

- class UpdateFulfilmentRate(APIView): Defines a new class named UpdateFulfilmentRate that inherits from APIView.

path('api/vendors/<int:vendor\_id>/performance/',VendorPerformanceEndpoint.as\_view()),

This is a URL pattern in Django that maps a URL to a view function. Here's a breakdown of the components:

- path('api/vendors/<int:vendor\_id>/performance/', VendorPerformanceEndpoint.as\_view())

- api/vendors/: This is the base URL path for the API endpoint.

- <int:vendor\_id>: This is a URL parameter that captures an integer value, which represents the ID of a vendor. The angle brackets <> indicate that this is a URL parameter.

- performance/: This is the final part of the URL path, which indicates that this endpoint is related to vendor performance.

- VendorPerformanceEndpoint.as\_view(): This is the view function that will handle requests to this URL. The as\_view() method is a class method provided by Django's View class, which returns a callable view function.

 def post(self,request):

*# Retrieve the vendor ID from the request data*

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

        try:

*# Retrieve the vendor and calculate fulfilment rate*

            vendor=Vendor.objects.annotate(

                total\_orders=Count('purchaseorder'),

                successful\_orders=Count('purchaseorder',filter=Q(purchaseorder\_\_status='completed',purchaseorder\_\_issues=0))

            ).filter(pk=vendor\_id).first()

            if vendor:

                fulfilment\_rate=(vendor.successful\_orders\*100)/vendor.total\_orders if vendor.total\_orders > 0 else 0

*# Update vendor's Fulfilment Rate*

                vendor.fulfilment\_rate=fulfilment\_rate

                vendor.save()

                return Response({'message':'Fulfilment Rate updated successfully.'})

            else:

                return Response({'error':'Vendor not found.'})

        except Exception as e:

            return Response({'error':str(e)})

This is a Django view function that handles a POST request to update a vendor's fulfillment rate. Here's a breakdown of what it does:

1. vendor\_id = request.data.get('vendor\_id'): Retrieves the vendor ID from the request data.

2. try:: Starts a try-except block to handle potential errors.

3. vendor = Vendor.objects.annotate(...).filter(pk=vendor\_id).first(): Retrieves the vendor instance with the specified ID, annotating it with the total number of orders and successful orders.

4. if vendor:: Checks if a vendor instance was found.

5. fulfilment\_rate = (vendor.successful\_orders \* 100) / vendor.total\_orders if vendor.total\_orders > 0 else 0: Calculates the fulfillment rate as a percentage.

6. vendor.fulfilment\_rate = fulfilment\_rate: Updates the vendor's fulfillment rate.

7. vendor.save(): Saves the updated vendor instance to the database.

8. return Response({'message': 'Fulfilment Rate updated successfully.'}): Returns a successful response message.

9. else: return Response({'error': 'Vendor not found.'}): If no vendor instance was found, returns an error response.

10. except Exception as e: return Response({'error': str(e)}): Catches any exceptions that occur during the execution of the view function and returns an error response with the exception message.

*# Data Integrity:*

class UpdateFulfilmentRate(APIView):

This view is likely designed to handle requests related to updating the fulfillment rate of a vendor or a purchase order.

Here's a breakdown of the class:

- class UpdateFulfilmentRate(APIView): Defines a new class named UpdateFulfilmentRate that inherits from APIView.

path('api/purchase\_orders/<int:po\_id>/acknowledge/',UpdateAcknowledgmentEndpoint.as\_view()),

This is a URL pattern in Django that maps a URL to a view function. Here's a breakdown of the components:

- api/purchase\_orders/: This is the base URL path for the API endpoint.

- <int:po\_id>: This is a URL parameter that captures an integer value, which represents the ID of a purchase order. The angle brackets <> indicate that this is a URL parameter.

- acknowledge/: This is the final part of the URL path, which indicates that this endpoint is related to acknowledging a purchase order.

- UpdateAcknowledgmentEndpoint.as\_view(): This is the view function that will handle requests to this URL. The as\_view() method is a class method provided by Django's View class, which returns a callable view function.

def post(self,request):

*# Retrieve the vendor ID from the request data*

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

        try:

*# Retrieve the vendor and calculate fulfilment rate*

            vendor=Vendor.objects.annotate(

                total\_orders=Count('purchaseorder'),

                successful\_orders=Count('purchaseorder',filter=Q(purchaseorder\_\_status='completed',purchaseorder\_\_issues=0))

            ).filter(pk=vendor\_id).first()

            if vendor:

                if vendor.total\_orders==0:

                    fulfilment\_rate=0

                else:

                    fulfilment\_rate=(vendor.successful\_orders\*100)/vendor.total\_orders

*# Update vendor's Fulfilment Rate*

                vendor.fulfilment\_rate=fulfilment\_rate

                vendor.save()

                return Response({'message':'Fulfilment Rate updated successfully.'})

            else:

                return Response({'error':'Vendor not found.'})

        except ZeroDivisionError:

            return Response({'error':'Cannot divide by zero.'})

        except Exception as e:

            return Response({'error':str(e)})

This is a Django view function that handles a POST request to update a vendor's fulfillment rate. Here's a breakdown of what it does:

1. vendor\_id = request.data.get('vendor\_id'): Retrieves the vendor ID from the request data.

2. try:: Starts a try-except block to handle potential errors.

3. vendor = Vendor.objects.annotate(...).filter(pk=vendor\_id).first(): Retrieves the vendor instance with the specified ID, annotating it with the total number of orders and successful orders.

4. if vendor:: Checks if a vendor instance was found.

5. if vendor.total\_orders == 0:: Checks if the vendor has any orders. If not, sets the fulfillment rate to 0.

6. fulfilment\_rate = (vendor.successful\_orders \* 100) / vendor.total\_orders: Calculates the fulfillment rate as a percentage.

7. vendor.fulfilment\_rate = fulfilment\_rate: Updates the vendor's fulfillment rate.

8. vendor.save(): Saves the updated vendor instance to the database.

9. return Response({'message': 'Fulfilment Rate updated successfully.'}): Returns a successful response message.

10. else: return Response({'error': 'Vendor not found.'}): If no vendor instance was found, returns an error response.

11. except ZeroDivisionError: return Response({'error': 'Cannot divide by zero.'}): Catches the ZeroDivisionError exception that occurs when trying to divide by zero and returns an error response.

12. except Exception as e: return Response({'error': str(e)}): Catches any other exceptions that occur during the execution of the view function and returns an error response with the exception message.

*# Real-time Updates:*

@receiver(post\_save,sender=PurchaseOrder)

def update\_fulfilment\_rate(sender, instance,\*\*kwargs):

    if instance.status=='completed' or instance.status=='cancelled':

*# Retrieve the vendor and update fulfilment rate*

        vendor=instance.vendor

        vendor.update\_fulfilment\_rate()

This is a Django signal receiver function that updates a vendor's fulfillment rate when a PurchaseOrder instance is saved. Here's a breakdown of what it does:

1. @receiver(post\_save, sender=PurchaseOrder): This is a decorator that registers the function as a receiver for the post\_save signal sent by the PurchaseOrder model.

2. def update\_fulfilment\_rate(sender, instance, \*\*kwargs):: This is the receiver function that will be called when the post\_save signal is sent.

3. if instance.status == 'completed' or instance.status == 'cancelled':: This checks if the saved PurchaseOrder instance has a status of either 'completed' or 'cancelled'.

4. vendor = instance.vendor: This retrieves the associated Vendor instance from the PurchaseOrder instance.

5. vendor.update\_fulfilment\_rate(): This calls the update\_fulfilment\_rate method on the Vendor instance to update its fulfillment rate.

 path('api/vendors/<int:vendor\_id>/update\_fulfilment\_rate/', UpdateFulfilmentRate.as\_view()),

This is a URL pattern in Django that maps a URL to a view function. Here's a breakdown of the components:

- api/vendors/: This is the base URL path for the API endpoint.

- <int:vendor\_id>: This is a URL parameter that captures an integer value, which represents the ID of a vendor. The angle brackets <> indicate that this is a URL parameter.

- update\_fulfilment\_rate/: This is the final part of the URL path, which indicates that this endpoint is used to update the fulfillment rate of a vendor.

- UpdateFulfilmentRate.as\_view(): This is the view function that will handle requests to this URL. The as\_view() method is a class method provided by Django's View class, which returns a callable view function.