Django\_views

Here is the implementation of CRUD operations for "Order Placement" Use Case.  
  
```python  
# Views.py  
  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
from.models import Product, Customer, Order  
from.serializers import ProductSerializer, CustomerSerializer, OrderSerializer  
  
  
class CreateOrder(APIView):  
 def post(self, request):  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order = Order(customer=customer)  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order created successfully'}, status=201)  
  
  
class GetOrders(APIView):  
 def get(self, request):  
 orders = Order.objects.all()  
 serializer = OrderSerializer(orders, many=True)  
 return Response(serializer.data)  
  
  
class DeleteOrder(APIView):  
 def delete(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 order.delete()  
 return Response({'message': 'Order deleted successfully'}, status=200)  
  
  
class UpdateOrder(APIView):  
 def put(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order.customer = customer  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order updated successfully'}, status=200)  
```  
  
This code provides four endpoints:  
  
\* `CreateOrder`: Creates a new order for a given customer and adds one or more products to that order.  
\* `GetOrders`: Retrieves all existing orders.  
\* `DeleteOrder`: Deletes an existing order.  
\* `UpdateOrder`: Updates an existing order by changing its associated customer and/or adding/removing products.  
  
The above code assumes that there exists a `Product` model and a `Customer` model, both having their respective serializers (`ProductSerializer`, `CustomerSerializer`). The `Order` model has a foreign key relationship with the `Customer` model, and also with the `OrderProduct` model through Many-To-Many relationships.  
  
For better understanding and debugging purposes, it would be helpful to create separate models for `OrderProduct` and include it as a nested JSON response within the main `Order` model. This will allow for easier tracking and management of individual order items.

Here is the implementation of CRUD operations for "Order Placement" Use Case.  
  
```python  
# Views.py  
  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
from.models import Product, Customer, Order  
from.serializers import ProductSerializer, CustomerSerializer, OrderSerializer  
  
  
class CreateOrder(APIView):  
 def post(self, request):  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order = Order(customer=customer)  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order created successfully'}, status=201)  
  
  
class GetOrders(APIView):  
 def get(self, request):  
 orders = Order.objects.all()  
 serializer = OrderSerializer(orders, many=True)  
 return Response(serializer.data)  
  
  
class DeleteOrder(APIView):  
 def delete(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 order.delete()  
 return Response({'message': 'Order deleted successfully'}, status=200)  
  
  
class UpdateOrder(APIView):  
 def put(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order.customer = customer  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order updated successfully'}, status=200)  
```  
  
This code provides four endpoints:  
  
\* `CreateOrder`: Creates a new order for a given customer and adds one or more products to that order.  
\* `GetOrders`: Retrieves all existing orders.  
\* `DeleteOrder`: Deletes an existing order.  
\* `UpdateOrder`: Updates an existing order by changing its associated customer and/or adding/removing products.  
  
The above code assumes that there exists a `Product` model and a `Customer` model, both having their respective serializers (`ProductSerializer`, `CustomerSerializer`). The `Order` model has a foreign key relationship with the `Customer` model, and also with the `OrderProduct` model through Many-To-Many relationships.  
  
For better understanding and debugging purposes, it would be helpful to create separate models for `OrderProduct` and include it as a nested JSON response within the main `Order` model. This will allow for easier tracking and management of individual order items.

Here is the implementation of CRUD operations for "Order Placement" Use Case.  
  
```python  
# Views.py  
  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
from.models import Product, Customer, Order  
from.serializers import ProductSerializer, CustomerSerializer, OrderSerializer  
  
  
class CreateOrder(APIView):  
 def post(self, request):  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order = Order(customer=customer)  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order created successfully'}, status=201)  
  
  
class GetOrders(APIView):  
 def get(self, request):  
 orders = Order.objects.all()  
 serializer = OrderSerializer(orders, many=True)  
 return Response(serializer.data)  
  
  
class DeleteOrder(APIView):  
 def delete(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 order.delete()  
 return Response({'message': 'Order deleted successfully'}, status=200)  
  
  
class UpdateOrder(APIView):  
 def put(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order.customer = customer  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order updated successfully'}, status=200)  
```  
  
This code provides four endpoints:  
  
\* `CreateOrder`: Creates a new order for a given customer and adds one or more products to that order.  
\* `GetOrders`: Retrieves all existing orders.  
\* `DeleteOrder`: Deletes an existing order.  
\* `UpdateOrder`: Updates an existing order by changing its associated customer and/or adding/removing products.  
  
The above code assumes that there exists a `Product` model and a `Customer` model, both having their respective serializers (`ProductSerializer`, `CustomerSerializer`). The `Order` model has a foreign key relationship with the `Customer` model, and also with the `OrderProduct` model through Many-To-Many relationships.  
  
For better understanding and debugging purposes, it would be helpful to create separate models for `OrderProduct` and include it as a nested JSON response within the main `Order` model. This will allow for easier tracking and management of individual order items.

Here is the implementation of CRUD operations for "Order Placement" Use Case.  
  
```python  
# Views.py  
  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
from.models import Product, Customer, Order  
from.serializers import ProductSerializer, CustomerSerializer, OrderSerializer  
  
  
class CreateOrder(APIView):  
 def post(self, request):  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order = Order(customer=customer)  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order created successfully'}, status=201)  
  
  
class GetOrders(APIView):  
 def get(self, request):  
 orders = Order.objects.all()  
 serializer = OrderSerializer(orders, many=True)  
 return Response(serializer.data)  
  
  
class DeleteOrder(APIView):  
 def delete(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 order.delete()  
 return Response({'message': 'Order deleted successfully'}, status=200)  
  
  
class UpdateOrder(APIView):  
 def put(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order.customer = customer  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order updated successfully'}, status=200)  
```  
  
This code provides four endpoints:  
  
\* `CreateOrder`: Creates a new order for a given customer and adds one or more products to that order.  
\* `GetOrders`: Retrieves all existing orders.  
\* `DeleteOrder`: Deletes an existing order.  
\* `UpdateOrder`: Updates an existing order by changing its associated customer and/or adding/removing products.  
  
The above code assumes that there exists a `Product` model and a `Customer` model, both having their respective serializers (`ProductSerializer`, `CustomerSerializer`). The `Order` model has a foreign key relationship with the `Customer` model, and also with the `OrderProduct` model through Many-To-Many relationships.  
  
For better understanding and debugging purposes, it would be helpful to create separate models for `OrderProduct` and include it as a nested JSON response within the main `Order` model. This will allow for easier tracking and management of individual order items.

Here is the implementation of CRUD operations for "Order Placement" Use Case.  
  
```python  
# Views.py  
  
from rest\_framework.response import Response  
from rest\_framework.views import APIView  
from.models import Product, Customer, Order  
from.serializers import ProductSerializer, CustomerSerializer, OrderSerializer  
  
  
class CreateOrder(APIView):  
 def post(self, request):  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order = Order(customer=customer)  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order created successfully'}, status=201)  
  
  
class GetOrders(APIView):  
 def get(self, request):  
 orders = Order.objects.all()  
 serializer = OrderSerializer(orders, many=True)  
 return Response(serializer.data)  
  
  
class DeleteOrder(APIView):  
 def delete(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 order.delete()  
 return Response({'message': 'Order deleted successfully'}, status=200)  
  
  
class UpdateOrder(APIView):  
 def put(self, request, pk):  
 try:  
 order = Order.objects.get(pk=pk)  
 except Order.DoesNotExist:  
 return Response({'error': 'Order does not exist'}, status=404)  
  
 customer\_id = request.data['customer\_id']  
 product\_ids = request.data.getlist('product\_ids')  
  
 try:  
 customer = Customer.objects.get(id=customer\_id)  
 except Customer.DoesNotExist:  
 return Response({'error': 'Customer does not exist'}, status=404)  
  
 order.customer = customer  
 order.save()  
  
 for product\_id in product\_ids:  
 try:  
 product = Product.objects.get(id=product\_id)  
 except Product.DoesNotExist:  
 return Response({'error': 'Product does not exist'}, status=404)  
  
 order\_product = OrderProduct(order=order, product=product)  
 order\_product.save()  
  
 return Response({'message': 'Order updated successfully'}, status=200)  
```  
  
This code provides four endpoints:  
  
\* `CreateOrder`: Creates a new order for a given customer and adds one or more products to that order.  
\* `GetOrders`: Retrieves all existing orders.  
\* `DeleteOrder`: Deletes an existing order.  
\* `UpdateOrder`: Updates an existing order by changing its associated customer and/or adding/removing products.  
  
The above code assumes that there exists a `Product` model and a `Customer` model, both having their respective serializers (`ProductSerializer`, `CustomerSerializer`). The `Order` model has a foreign key relationship with the `Customer` model, and also with the `OrderProduct` model through Many-To-Many relationships.  
  
For better understanding and debugging purposes, it would be helpful to create separate models for `OrderProduct` and include it as a nested JSON response within the main `Order` model. This will allow for easier tracking and management of individual order items.