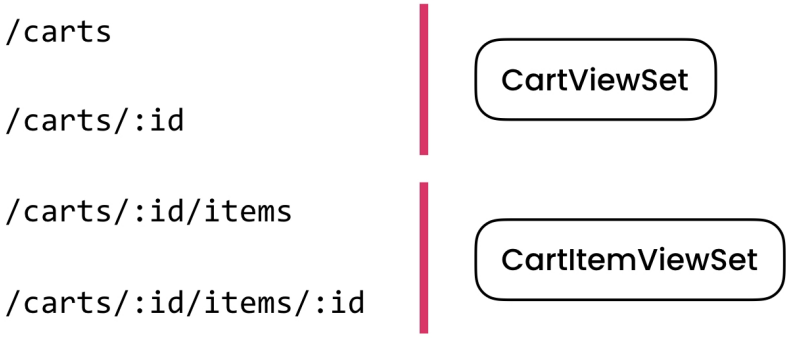
# Shopping Cart API Design & Implementation

Designing a Shopping Cart API

| Operation | Method | Body | Endpoint |
|---|---|---|---|
| Create a cart | `POST` | - | `/carts` |
| Add items to a cart | `POST` | `{product, quantity}` | `/carts/{pk}/items` |
| Get a item in a cart | `GET` | - | `/carts/{pk}/items/{pk}` |
| Update the quantity of items | `PATCH` | `{quantity}` | `/carts/{pk}/items/{pk}` |
| Remove items from a cart | `DELETE` | - | `/carts/{pk}/items/{pk}` |
| Get a cart with its items | `GET` | - | `/carts/{pk}` |
| Delete a cart | `DELETE` | - | `/carts/{pk}` |




Revisiting the Data Model

The ID for cart model is automatically set by Django (integer) which is not a good approach when dealing with sensitive information.

- It can be guessed. meaning people can alter carts that are not theirs

# Solution

- using GUIDs (Globally Unique Identifier) as primary key- 32 character string
- using integers as Primary Key and using GUIDs only in API layer

# Using a GUID

```
from uuid import uuid4

class Cart(models.Model):
        id = models.UUIDField(primary_key=True,
default=uuid4)
        created_at =
models.DateTimeField(auto_now_add=True)
```

the implication of this change

- `bigint` (8 bytes) to `guid` (32 bytes) in the `cart` table
- the same change in `cartitem` table for foreign key to `cart`

> ***Is this a problem?***

Depends... some people may say yes. because it slows our performance. But Logically speaking:

- 1,000,000 * 24 = 24,000,000 -> Extra Space in bytes
- (24,000,000 / 1024) / 1024 = 24 -> MB
- This table is temporary
- Disk Space is pretty cheap

as people place orders, these records are going to be transferred to `Order` and `OrderItem` tables. In those tables we don't use GUIDs because the orders API is going to be secure and not open to public.

# Lookup Performance

In theory, looking up a GUID field is slower than looking up integers. But these days servers are powerful enough that the difference is not that big. Also popular DBMS are highly optimized.

> Don't optimize before testing and experiencing problems
>
> ***Premature Optimization is the root of all evil***

---

```
class Cart(models.Model):
   id = models.UUIDField(primary_key=True, default=uuid4())
   created_at = models.DateTimeField(auto_now_add=True)
```

```
class CartItem(models.Model):
    cart = models.ForeignKey(Cart,
on_delete=models.CASCADE, related_name='items')
    product = models.ForeignKey(Product,
on_delete=models.CASCADE)
    quantity = models.PositiveSmallIntegerField()

    class Meta:
        unique_together = [['cart', 'product']]
```

Creating a Cart

Steps

1. Create a `Serializer`
2. Create a `ViewSet`
3. Register a `Route`

---

## `Serializer`

```python
class CartSerializer(serializers.ModelSerializer):
    class Meta:
        model = Cart
        fields = [
            'id'
        ]
        read_only_fields = ['id']
```

---

## ViewSet

```python
class CartViewSet(GenericViewSet, CreateModelMixin):
        queryset = Cart.objects.all()
        serializer_class = CartSerializer
```

---

## Route

```python
router.register('carts', views.CartViewSet)
```

Getting a Cart

- We should get a cart with all its items

like this:

```
{
        "item": {
                "product": {
                        ...
                },
                "total_price": 10,
        },
        "total_price": 10
}
```

## A simple representation of `Products`

```
class SimpleProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = [
            'id',
            'title',
            'unit_price'
        ]
```

## `CartItemSerializer`

```
class CartItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = CartItem
        fields = [
            'id',
            'product',
            'quantity',
            'total_price'
        ]

    product = SimpleProductSerializer()
    total_price = serializers.SerializerMethodField()

    @staticmethod
    def get_total_price(cart_item: CartItem):
        return cart_item.quantity *
cart_item.product.unit_price
```

## `CartSerializer`

```
class CartSerializer(serializers.ModelSerializer):
    class Meta:
        model = Cart
        fields = [
            'id',
            'items',
```

```
            'total_price'
        ]
        read_only_fields = ['id']

    items = CartItemSerializer(many=True, read_only=True)
    total_price =
serializers.SerializerMethodField(read_only=True)

    @staticmethod
    def get_total_price(cart: Cart):
        return sum(
                [
                        item.quantity *
item.product.unit_price
                        for item in cart.items.all()
                ]
        )
```

After Checking performance

```
class CartViewSet(GenericViewSet,
                  CreateModelMixin,
                  RetrieveModelMixin,
                  DestroyModelMixin):
    queryset =
Cart.objects.prefetch_related('items__product').all() #!
    serializer_class = CartSerializer
```

Deleting a Cart

- Add `DestoryModelMixin` to achieve delete behavior

```
class CartViewSet(GenericViewSet,
                  CreateModelMixin,
                  RetrieveModelMixin,
                  DestroyModelMixin):
    queryset =
Cart.objects.prefetch_related('items__product').all()
    serializer_class = CartSerializer
```

Getting & Adding Cart Items

`/carts/uuid/items` - must show all cart items (With `SimpleProductSerializer`)
The only difference between this endpoint and Cart endpoint is that we don't have Cart specific properties like:

- Cart ID
- Total Cart Price

```
[
        {cartitem},
        {cartitem},
        ...
]
```

---

New `ViewSet`

```python
class CartItemViewSet(ModelViewSet):

    def get_queryset(self):
        return (
            CartItem
            .objects
            .select_related('product')
            .filter(cart_id=self.kwargs['cart_pk'])
        )

    def get_serializer_class(self):
        if self.request.method == 'POST':
            return AddCartItemSerializer
        return CartItemSerializer

    def get_serializer_context(self):
        return {'cart_id': self.kwargs['cart_pk']}
```

# Add Serializer

```
class AddCartItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = CartItem
        fields = [
            'id',
            'product_id',
            'quantity'
        ]

    product_id = serializers.IntegerField()

    @staticmethod
    def validate_product_id(value):
        if not Product.objects.filter(pk=value).exists():
            raise serializers.ValidationError("...")
        return value

    def save(self, **kwargs):
        cart_id = self.context['cart_id']
        product_id = self.validated_data['product_id']
        quantity = self.validated_data['quantity']
        try:
            cart_item = (
                    CartItem
                    .objects
                    .get(cart_id=cart_id,
product_id=product_id)
            )
            cart_item.quantity += quantity
            cart_item.save()
            self.instance = cart_item
        except CartItem.DoesNotExist:
            self.instance = (
                    CartItem
                    .objects
                    .create(cart_id=cart_id,
**self.validated_data)
```

```
                )
        return self.instance
```

## Route

```
router.register('carts', views.CartViewSet)

carts_router = routers.NestedDefaultRouter(
        router,
        'carts',
        lookup='cart'
)
carts_router.register(
        'items',
         views.CartItemViewSet,
         basename='cart-items'
)
```

Updating a CartItem

We only want to update the `quantity` -> `PATCH`

- Create a Custom Serializer for updating
- Put the required conditionals in `get_serializer_class()`

---

```
class
UpdateCartItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = CartItem
        fields = [
            'quantity'
        ]
```

## `ViewSet` Changes

```python
class CartItemViewSet(ModelViewSet):
    http_method_names = [
        'options',
        'get',
        'post',
        'patch',
        'delete'
    ]

    def get_queryset(self):
        return (
            CartItem
            .objects
            .select_related('product')
            .filter(cart_id=self.kwargs['cart_pk'])
        )

    def get_serializer_class(self):
        if self.request.method == 'POST':
            return AddCartItemSerializer
        if self.request.method == 'PATCH':
            return UpdateCartItemSerializer
        return CartItemSerializer

    def get_serializer_context(self):
        return {'cart_id': self.kwargs['cart_pk']}
```

Deleting a CartItem

Already Implemented