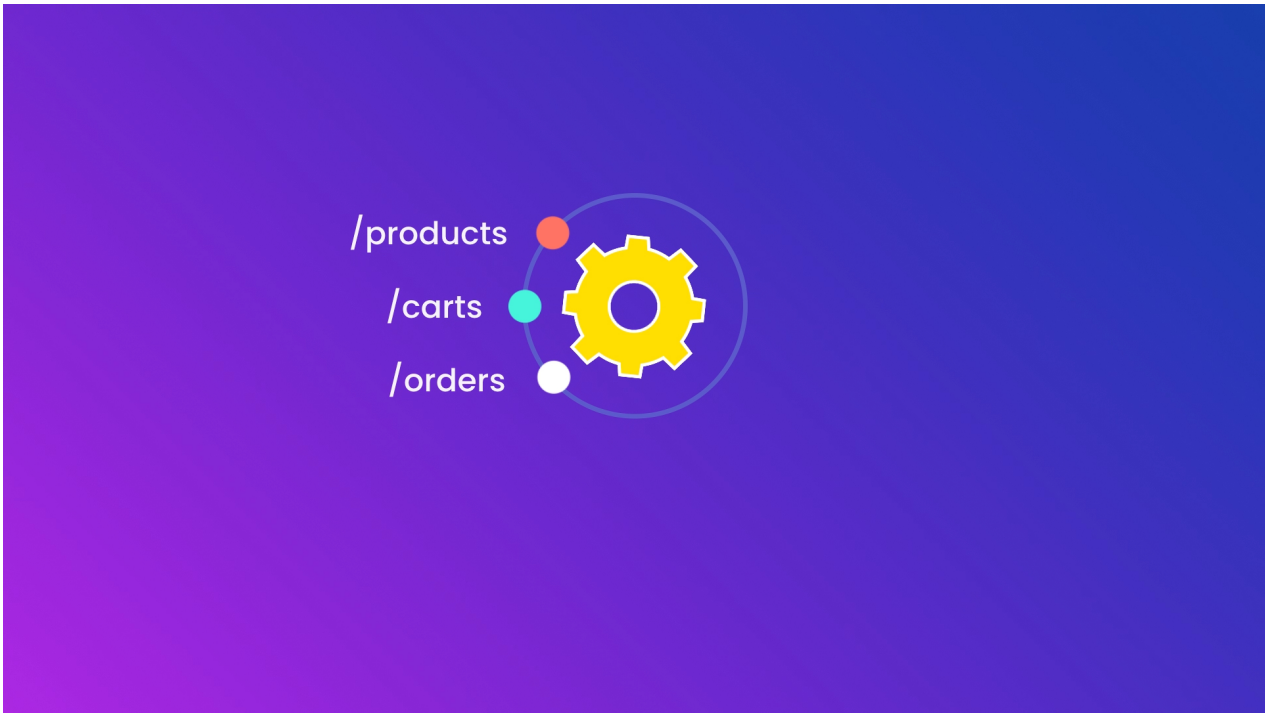
Building RESTful APIs

> What are RESTful APIs
>
> - So far the only method for manipulating the data of our application is through the **Admin Panel**
> - We need a way for client applications to **CRUD** our data



What makes an API RESTful?

- **RE**presentational
- **S**tate
- **T**ransfer

Practical Terms

Rest Defines rules for clients and servers to communicate with each other over the web
Following these rules results in our systems being:

- Fast
- Scalable

- Reliable
- Easy to understand
- Easy to change

> An API that confirms to these rules is called **RESTful**

What are these rules

This is a highly complex topic but in short:

- Resources
- Resource Representations
- HTTP Methods

> Resources
>
> A resource in an API is like an object in our application
>
> - Product
> - Collection
> - Cart
>
> These resources need to be available on the web and client applications can access them using a **URL**.
>
> - List of products - `domain/products`
> - Individual product - `domain/products/1`
> - Nested resources - `domain/products/1/reviews`
> - Nested resources - `domain/products/1/reviews/1`
>   Nesting Resources too deep results in problems anything beyond 2 levels is unnecessary.
>
> As you can see there's a consistency and pattern across our URLs. This is one the attributes of **RESTful APIs**. if we follow this pattern, our API will be familiar and easy to understand for others.

> Resource Representations

When we Request a resource, the server is going to return the response in a special format. Some of these formats are:

- HTML - Like templates
- XML - Not used anymore
- JSON - Most popular

> Important
>
> None of these are the internal representation of our resources (On the Server)
>
> - Internal Representation - OOP
> - External Representation - JSON

if we support more than one representation, the client needs to tell the server what format it wants the data to be in.

JSON

JavaScript Object Notation

```
{
 "name": "john",
 "age": 32,
 "is_online": true,
 "employer": {},
 "interests": []
}
```

HTTP Methods

- GET - Reading Data
- POST - Writing Data
- PUT - Updating Data
- PATCH - Partially updating Data

- DELETE - Delete Data

Endpoints Support various operations. Some may only allow Reading Data (GET), While others have operations based on permission.

> Using HTTP Methods the client can tell the server what it wants to do with the resource.

Scenarios

- Creating a Product - `POST /products`
  - Data (Body) - `{"title": "Orange Juice", "price": 19.35, ...}`
- Updating a Product
  - Updating all Properties - `PUT /products/1`
  - Updating some Properties - `PATCH /products/1`
  - Data (Body) - `{title: 'Apple juice', ...}`
- Deleting a Product
  - `DELETE /products/1`
  - No Body. the only data needed is the ID which is provided in the URL

> Installing Django REST Framework
>
> - This is Separate Framework that sits on top of Django
> - DRF Makes Building RESTful APIs very easy

```
pipenv install djangorestframework
```

```
INSTALLED_APPS = [
```

```
    ...
    'rest_framework',
]
```

Creating API Views

- Creating an endpoint like `/store/products`
    - See the all products in the database

List View

```
# Store/views.py
from django.http import HttpResponse

def product_list(request):
        return HttpResponse('ok') # Keeping it simple
```

Mapping view to URL

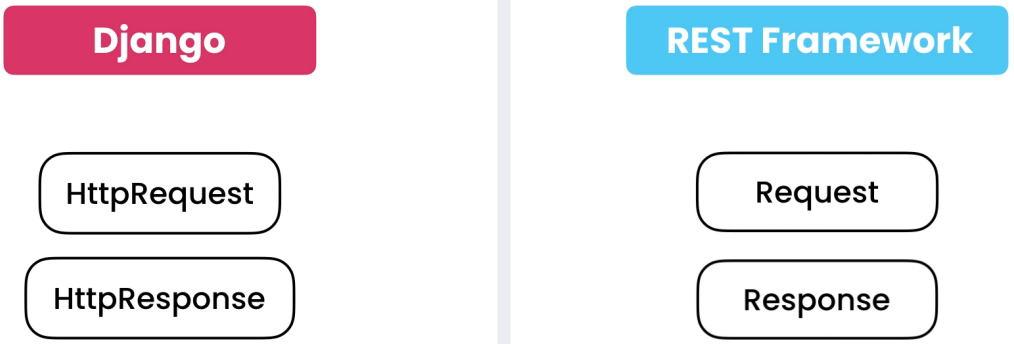
```
# store/urls.py
from . import views

urlpatterns = [
        path('products/', views.product_list)
]
```

Root URL

```
urlpatterns = [
        path('store/', include('store.urls'))
]
```

So far this is how we create a view in Django, nothing new...



These classes are simpler and more powerful than the ones that come with Django

```
# store/views.py
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view() # → Request Instance is going to be the one in
DRF
def product_list(request):
        return Response('ok') # → Response Object in DRF
```

Now if we hit the endpoint, we get the **Browsable API** page

- Only Visible in Browser
- Client app only gets the data

Browsable API Sections

- Heading - Generated Based on Endpoint
- Request Information - `GET /store/products`
- Response
    - HTTP status code
    - Allow - allowed operations

- content-type - type of content in the response
- Vary: Caching related

Detail View

```
@api_view
def product_details(request, id):
        return Response(id)
```

Map to URL

```
urlpatterns = [
        path('products/<int:id>/', views.product_detail)
]
```

> ⚡ Non-numeric ID
>
> If we pass a non-numeric id to this parameter, we are going to get an Logical error. this can be countered with Convertors. After that `<int:id>` if any non-numeric values are passed we get an error.

Creating Serializers

We need a way to convert **OOP** objects to **JSON** objects

- in DRF we have a class named `JSONRenderer`
  - which has a `render()` method that takes a python dictionary as argument and returns a JSON object.

So we need a way to convert OOP instances to Dictionaries and pass it to this method and receive a JSON object.

> This is where **Serializers** come into the picture

Serializer

an object that knows how to convert a model instance into a dictionary

Writing a Serializer

```
# Create new module <<serializers.py>>
from rest_framework import serializers

class ProductSerializer(serializers.Serializer):
        id = serializers.IntegerField()
        title = serializers.CharField(max_length=255)
        price = serializers.DecimalField( # names can be
different
                max_digits=6,
                decimal_places=2
        )
```

- What we return from our API doesn't necessarily need to have all the Model fields - Internal Representation(Model) can be different from the External(API) one
- Sometimes we might have sensitive information in the internal representation

Serializing Objects

```
# store/views
from .serializers import ProductSerializer

@api_view()
def product_detail(request, id):
        product = Product.objects.get(pk=id)
        # Serialization happens here
        serializer = ProductSerializer(product)
        return Response(serializer.data) # to access the
dict
```

Response

```
{
    "id": 1,
    "title": "Bread Ww Cluster",
    "unit_price": "4.00"
}
```

as you can see the `unit_price` field is rendered as a string even though we said it was a Decimal Field in the serializer Definition. This is one of the default settings in DRF which can be easily changed.

```
# settings module
REST_FRAMEWORK = {
        'COERCE_DECIMAL_TO_STRING': False
}
```

Asking for a product that does not exist

- `GET /store/products/0` - Throws an exception

> ⚡ RESTful Convention
>
> One of the rules of REST is that if an object does not exist we should return a response and the status of the response should be `404` which in HTTP means **Not Found**

Solution

- We can use a `try/except` block to counter this. But repeating this pattern is time consuming
- The optimal solution is to use `get_object_or_404`
    - This function wraps the `try/except` block

```
from django.shortcuts import get_object_or_404

@api_view()
def product_detail(request, pk):
        product = get_object_or_404(Product, pk=pk)
        serializer = ProductSerializer(product)
        return Response(serializer.data)
```

Now let's modify the `product_list` view

```
@api_view()
def product_list(request):
        queryset = Product.objects.all()
        serializer = ProductSerializer(queryset, many=True)
        return Response(serializer.data)
```

Custom Serializer Fields

- The objects we return from our API, don't necessarily need to be Like the objects we have in our Data Model. Because our Data Models are really implementation details and may change in the future. we don't want these changes to be exposed to the outside world. *Remote Control example*
- In contrast, our API is the interface to our application and frequent changes results in existing clients breaking.
    - Usually changes through APIs is done by making different versions

---

Just how we excluded existing fields from our model in our API representation, we can add fields that don't exist in the Data model and only in the External Representation.

```
from decimal import Decimal

class ProductSerializer(serializers.Serializer):
```

```
        price_with_tax = serialziers.SerializerMethodField(
                method_name='calculate_tax'
        )

        def calculate_tax(self, product: Product):
                return product.unit_price * Decimal(1.1)
```

Renaming Fields

```
price = serializer.DecimalField(..., source='unit_price')
```

Serializing Relationships

- Show a product with its collection
  - return the id only
  - return collection as a string (title)
  - return the full collection object in the product object (Nesting)
  - return a hyperlink to the collection endpoint

ID only

```
class ProductSerializer(serializer.Serializer):
        collection = serializers.PrimaryKeyRelatedField(
                queryset=Collection.objects.all()
        )
```

String

```
class ProductSerializer(serializer.Serializer):
        collection = serializer.StringRelatedField()
        # Eager loading is required in view
```

```
# Eager Loading
@api_view()
def product_list(request):
        queryset =
Product.objects.select_related('collection').all()
```

Nesting

- First we create a Collection Serializer
- Then we include it in the Product Serializer

```
class CollectionSerializer(serializers.Serializer):
        id = serializer.IntegerField()
        title = serializers.CharField(max_length=255)

class ProductSerializer(serializer.Serializer):
        collection = CollectionSerializer()
```

Now each collection is rendered as a JSON object inside each Product

---

Hyperlink

```
class ProductSerializer(serializer.Serializer):
        collection = serializers.HyperlinkRelatedField(
                queryset=Collection.objects.all(),
                view_name='collection-detail' # define in
urls module
        )
```

```
# local urls module
urlpatterns = [
        path(
                'collections/<int:id>',
                 views.collection_detail,
```

```
                name='collection-detail'
        )
]
```

```
# views module
@api_view
def collection_detail(request, pk):
        collection = get_object_or_404(Collection, pk=pk)
        serializer = CollectionSerializer(collection)
        return Response(serializer.data)
```

⚡ Assertion Error

we need to add a context object to the serializer

```
def products_list(request):
        queryset = ...
        serializer = ProductSerializer(
                ..., context={'request': request}
        )
```

⚡ Exception

just use `pk` instead of `id` in URL patterns

Model Serializers

- a smarter way to make serializers

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

```
                        'price_with_tax',
                        'collection'
                ]
        collection = CollectionSerializer()
        price_with_tax = ...
```

Deserializing Data

This is the opposite of serialization and it happens when we receive data from the client.

Creating Product

Request: `POST /products`
Body:

```
{
        "title": "Apple Juice",
        "price": 10
}
```

so in order to save this to the database, first we have to present this data as a **Product Object** which means we need to **Deserialize** it.

```
@api_view(['GET', 'POST'])
def product_list(request):
    if request.method == "GET":
        queryset = (
                Product
                .objects
                .select_related('collection')
                .all()
        )
        serializer = ProductSerializer(queryset, many=True)
        return Response(serializer.data)
    elif request.method == "POST":
        serializer = ProductSerializer(data=request.data)
        return Response("OK")
```

Something Cool >

After adding the `POST` option in allowed methods, a form get's added to the **Browsable API**.

Data Validation

In order to use the `serializer.validated_data`, we need to validate the data using `serializer.is_valid()`. otherwise we get an exception

Exception Info

> You Must call `.is_valid()` before accessing `.validated_data`

```
if serializer.is_valid():
    serializer.validated_data
    return Response("OK")
else:
    return Response(
        serializer.errors,
        status=status.HTTP_400_BAD_REQUEST
    )
```

Response in case of posting empty data

```
{
    "title": [
        "This field is required."
    ],
    "unit_price": [
        "This field is required."
    ],
    "collection": [
        "This field is required."
    ]
}
```

A better way

```
elif request.method == "POST":
    serializer = ProductSerializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    serializer.validated_data
    return Response("OK")
```

Does the same thing with less code

Validated Data

This is how the Data looks after validation

```
{
        'title': 'Apple Juice',
        'unit_price': Decimal('10.59'),
        'collection': <Collection: collection1>
}
```

as you can see as part of Deserializing the data, DRF automatically retrieved a collection with the ID that we specified.

Validation at the object level

If we need anything more than the default validation, we have to override the validate method in our serializer.

```
class Serializer(ModelSerializer):
        ...
        def validate(self, data: dict):
                if data['password'] ≠
data['confirm_password']:
                        return serializers.ValidationError(
```

```
                                    "Passwords Don't match"
                                )
                return data
```

Saving Objects

Model Serializer has a `save()` method for creating and updating a product.

```
serializer.is_valid(raise_exception=True)
serializer.save()
```

> Tip >
>
> Change the Product Serializer Implementation based on the model and required fields

There are situations where we need to alter the default implementation of an object's creation.

```
def create(self, validated_data: dict):
        product = Product(**validated_data)
        product.other = 1
        product.save()
        return product

# For updating
def update(self, instance, validated_data):
        instance.unit_price =
validated_data.get('unit_price')
        instance.save()
        return instance
```

> ***The `save()` method will call one these methods depending on the state of the serializer***

Updating

For updating, we should modify the `product_detail` view. Because the `PUT` and `PATCH` requests are sent to a specific object like `/products/1`

```
@api_view(['GET', 'PUT'])
def product_detail(request, pk: int):
    product = get_object_or_404(Product, pk=pk)
    if request.method == 'GET':
        serializer = ProductSerializer(product)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = ProductSerializer(product,
data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data,
status=status.HTTP_200_OK)
```

Deleting Objects

```
elif request.method == 'DELETE':
    product.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)
```

🐞 Protected Foreign Key Error

we cannot delete products that are referenced by order items. so this situation calls for overriding the default implementation. We don't want to show an exception to the user, instead we should prompt the user with the correct message.

```
elif request.method == 'DELETE':
    if product.orderitems.count() > 0:
        return Response(
            {"error": "Product cannot be deleted because
it's associated with an order item."},
```

```
            status=status.HTTP_405_METHOD_NOT_ALLOWED
        )
    product.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)
```

Exercise

> ***Build The Collections API***