

DJANGO REST FRAMEWORK

CS 3030: Python

Instructor: Damià Fuentes Escoté



University of Colorado
Colorado Springs

Quiz 4

Project

- Your project ideas are awesome!
- For your specific project try to think more on python stuff (backend, API, python library – PyPi, etc) rather than creating a GUI webpage interface.
- You should start working!
 - ***Make sure that from the beginning you use GitHub.***

Web frameworks

- Django, Flask, Pyramid, Tornado, Bottle, Diesel, Pecan, Falcon, etc
- As a developer you want to cut the legions of options down to the one that will help you finish your project and get on to the Next Big Thing.

Web frameworks

Most famous and well documented

- Django, Flask, Pyramid, Tornado, Bottle, Diesel, Pecan, Falcon, etc
- As a developer you want to cut the legions of options down to the one that will help you finish your project and get on to the Next Big Thing.

Web frameworks

Most famous and well documented

- Django, Flask, Pyramid, Tornado, Bottle, Diesel, Pecan, Falcon, etc
- As a developer you want to cut the legions of options down to the one that will help you finish your project and get on to the Next Big Thing.
- Tutorial to build the same apps in Django, Flask and Pyramid and help you choose:
 - <https://www.airpair.com/python/posts/django-flask-pyramid>

Web frameworks

- Flask is a "microframework" primarily aimed at small applications with simpler requirements.
- Pyramid and Django are both aimed at larger applications, but take different approaches to extensibility and flexibility.
- Pyramid targets flexibility and lets the developer use the right tools for their project. This means the developer can choose the database, URL structure, templating style, and more.
- Django aims to include all the batteries a web application will need so developers need only open the box and start working, pulling in Django's many modules as they go.

Understanding REST

- REST (Representational State Transfer)
- REST is an architectural style for designing distributed systems. It is not a standard but a set of constraints, such as being stateless, having a client/server relationship, and a uniform interface.

Principles of REST

■ Resources

- *Expose easily understood directory structure URIs.*

■ Representations

- *Transfer JSON or XML to represent data objects and attributes.*

■ Messages

- *Use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).*

■ Stateless

- *Interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.*

HTTP methods

- Use HTTP methods to map CRUD (create, retrieve, update, delete) operations to HTTP requests.
 - *GET*
 - *POST*
 - *PUT*
 - *PATCH*
 - *DELETE*

GET

- Retrieve information.
- GET requests must be safe and **idempotent**, meaning regardless of how many times it repeats with the same parameters, the results are the same.

POST

- Request that the resource at the URI do something with the provided entity.
- Often POST is used to create a new entity, but it can also be used to update an entity.

PUT

- Store an entity at a URI. PUT can create a new entity or update an existing one.
- A PUT request is **idempotent**. Idempotency is the main difference between the expectations of PUT versus a POST request. So if you PUT an object twice, it has no effect.
 - This is a nice property, so I would use PUT when possible.
 - Drawback: The client needs to provide the Primary Key for the object in order to be idempotent.

PATCH


- Update only the specified fields of an entity at a URI.
- A PATCH request is neither safe nor idempotent. That's because a PATCH operation cannot ensure the entire resource has been updated.

DELETE


- Request that a resource be removed.
- However, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.

HTTP status codes

- Status codes indicate the result of the HTTP request:
 - *1XX - informational*
 - *2XX – success*
 - 201 Created
 - 200 OK
 - *3XX - redirection*
 - *4XX - client error*
 - 404 Not found
 - 405 Method not allowed
 - 409 Conflict
 - *5XX - server error*



DJANGO REST FRAMEWORK



Django REST framework

- Django REST framework is a powerful and flexible toolkit for building Web APIs.
- Traditionally, Django is known to many developers as a MVC Web Framework, but it can also be used to build a backend, which in this case is an API.
- DRF = Django REST Framework

What are we going to create?

- A very simple social network similar to a basic Instagram REST api.
 - *Users and authentication*
 - *Post images*
 - *Get list of users and list of posts.*

Overview – Homework/tutorial

1. **Serialization**
2. **Requests and responses**
3. **Class based views**
4. **Authentication and permissions**
5. **Relationships and hyperlinked APIs**
6. **Handling images**
7. **Upload your server to the internet (not required for the homework)**

- The tutorial/homework is based in a tutorial in the DRF webpage:
 - *<https://www.django-rest-framework.org>*

**How are
we going
to work
with DRF?**

Models

Serializers

Generic Views

Urls

Models

- A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

```
from django.db import models
```

```
class Person(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=30)
```

Serializers

- Converts the information stored in the database and defined by the Django models into a format which is more easily transmitted via an API, like JSON, XML, etc.
- Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.
- The DRF serializer handles this translation. When a user submits information (such as creating a new instance) through the API, the serializer takes the data, validates it, and converts it into something Django can slot into a Model instance. Similarly, when a user accesses information via the API the relevant instances are fed into the serializer, which parses them into a format that can easily be fed out as JSON to the user.

```
class PersonSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        model = Person
```

```
        fields = ('first_name', 'last_name', )
```

Generics Views

- A given serializer will parse information in both directions (reads and writes), but the View is where the available operations are defined.
- Using the generic APIView we can easily build methods that will list, create, retrieve, update and delete instances of our models. All through the help of the serializer of a specific model.

```
class PersonList(generics.ListCreateAPIView):  
    queryset = Person.objects.all()  
    serializer_class = UserSerializer  
    permission_classes = (IsAdminUser,)
```


Generics Views

■ CreateAPIView	—	post
■ ListAPIView	—	get
■ RetrieveAPIView	—	get
■ DestroyAPIView	—	delete
■ UpdateAPIView	—	put, patch
■ ListCreateAPIView	—	get, post
■ RetrieveUpdateAPIView	—	get, put, patch
■ RetrieveDestroyAPIView	—	get, delete
■ RetrieveUpdateDestroyAPIView	—	get, put, patch, delete

URLs

- How do we link our Views with URLs accessible from the outside. The surface layer.

```
urlpatterns = [  
    path('users/', views.PersonList.as_view(), name="person-list"),  
    path('user/<int:pk>/', views.PersonDetail.as_view(), name="person-detail"),]
```

DRF has even more abstraction

- DRF have more abstraction than the one we are going to see:
 - **ViewSets**
Automatically defines the functions (read, create, update, delete) which will be available via the API.
 - **Routers**
Automatically defines the URLs which will provide access to each viewset so all of them are defined in the same way.

- <https://www.caktusgroup.com/blog/2018/02/26/basics-django-rest-framework/>

How to handle images

- It is not recommended to save images in web applications servers. In that case the images will be stored in your database. There are a few problems with storing files on a database - files are generally much larger than your average row. **A database containing many large files will consume a lot of memory.**
- What it is done is to save the images to, for example, Azure blob storage server and then in our RESTful server database we only save a string reference of the the image. In the client side, with that reference we can access to the storage server and retrieve the photo.

How to handle images

- The process for uploading an image would be:
 1. Post the image to our DRF server
 2. DRF will choose a filename for that file and will post the image to Azure Blob storage
 3. DRF will respond to the client with the filename
 4. The client will receive the filename and will update the instance with the new filename.