@ How to create **a Django project on Eclipse.** < **File - New - PyDev Django Project** >

- **virtualenv** to install all files in the requirements.txt file

D:\venv37>**virtualenv -p** D:\Python37-32\python.exe **D:\venv37**

CD to D:\venv37
activate your **virtualenv** (Scripts\activate)
pip install -r D:\venv37\requirements.txt

https://medium.com/quick-code/token-based-authentication-for-django-rest-framework-44586a9a56fb

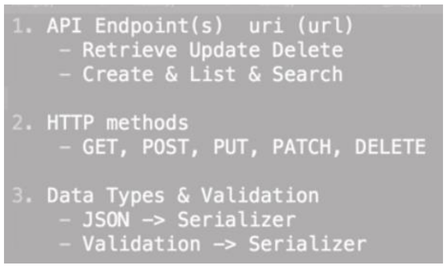
1, BasicAuthentication
2, TokenAuthentication
3, SessionAuthentication
4, RemoteUserAuthentication

these are all provided by drf(django rest framework) and other than these like oauth, oauth2 based authentication are provided by the efforts of the community with help of other python packages.
And they can be easily used in the production environment.

We gonna follow these steps

1, Installing Django
2, Making the Django Project
3, Installing Django Rest Framework
4, Setup the Login Function and api route
5, Testing the Login api route
6, Using the Token to access authenticated Api

```
D:\venv37>Scripts\activate

(venv37) D:\venv37>pip freeze
(venv37) D:\venv37>pip install Django
(venv37) D:\venv37>django-admin startproject myproject
    // this will create default app named "myproject" same as the project name.
(venv37) D:\venv37>pip install djangorestframework

// on eclipse: Open projects from file system...
// on eclipse: Window/preference/pydev/interpreters/python interpreter/new    <-- add venv36
// on eclipse: Right click of Project/Run As/Run Configurations/ type "runserver" on argument tab

// Update the settings.py file for the rest framework and token based authentication
// In the myproject folder create a views.py
// make api route for the same in the urls.py

(venv37) D:\venv37>cd myproject
    // if you want another app like "myapp2" ==> (venv37) D:\venv37\myproject>python manage.py startapp myapp2
(venv37) D:\venv37\myproject>python manage.py makemigrations
(venv37) D:\venv37\myproject>python manage.py migrate

(venv37) D:\venv37\myproject>python manage.py createsuperuser
kyung.lee\xx??xxxx
(venv37) D:\venv37\myproject>python manage.py runserver
        or You can run on eclipse.
```



```
1. API Endpoint(s)  uri (url)
     - Retrieve Update Delete
     - Create & List & Search

2. HTTP methods
     - GET, POST, PUT, PATCH, DELETE

3. Data Types & Validation
     - JSON -> Serializer
     - Validation -> Serializer
```

API Guide

Serializers allow complex data such as **querysets** and **model instances** to be converted to native Python datatypes that can then be easily rendered into `JSON`, `XML` or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The serializers in REST framework work very similarly to Django's `Form` and `ModelForm` classes. We provide a `Serializer` class which gives you a powerful, generic way to control the output of your responses, as well as a `ModelSerializer` class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

@ differentiate null=True, blank=True in django

```
models.CharField(max_length=100, null=True, blank=True)
```

`null=True` sets `NULL` (versus `NOT NULL`) on the column in your DB.
Blank values for Django field types such as `DateTimeField` or `ForeignKey` will be stored as `NULL` in the DB.

`blank=True` determines whether the field will be required in forms. This includes the admin and your own custom forms. If `blank=True` then the field will not be required, whereas if it's `False` the field cannot be blank.

@ Routers

REST framework adds support for **automatic URL routing** to Django, and provides you with a simple, quick and consistent way of wiring your view logic to a set of URLs.

Here's an example of a simple URL conf, that uses `SimpleRouter`.

```
from rest_framework import routers

router = routers.SimpleRouter()
router.register(r'users', UserViewSet)
router.register(r'accounts', AccountViewSet)
urlpatterns = router.urls
```

There are two mandatory arguments to the `register()` method:

- `prefix` - The URL prefix to use for this set of routes.
- `viewset` - The viewset class.

Optionally, you may also specify an additional argument:

- `basename` - The base to use for the URL names that are created. If unset the basename will be automatically generated based on the `queryset` attribute of the viewset, if it has one. Note that if the viewset does not include a `queryset` attribute then you must set `basename` when registering the viewset.

Typically, you won't *need* to specify the `basename` argument, but if you have a viewset where you've defined a custom `get_queryset` method, then the viewset may not have a `.queryset` attribute set.

generics vs viewset in django rest framework, how to prefer which one to use?

DRF has two main systems for handling views:

1. APIView: This provides some **handler methods**, to handle the http verbs: **`get, post, put, patch`**, and **`delete`**.

2. ViewSet: This is an abstraction over APIView, which provides **actions** as **methods**:
   - **`list`**: read only, returns multiple resources (http verb: `get`). Returns a list of dicts.
   - **`retrieve`**: read only, single resource (http verb: `get`, but will expect an id). Returns a single dict.
   - **`create`**: creates a new resource (http verb: `post`)
   - **`update/partial_update`**: edits a resource (http verbs: `put/patch`)
   - **`destroy`**: removes a resource (http verb: `delete`)

Both can be used with normal django urls.

Because of the conventions established with the **actions**, the `ViewSet` has also the ability to be mapped into a **router**, which is really helpful.
Now, both of this Views, have **shortcuts**, these shortcuts give you a simple implementation ready to be used.

GenericAPIView: for `APIView`, this gives you shortcuts that map closely to your database models. Adds commonly required behavior for standard list and detail views. Gives you some attributes like, the **`serializer_class`**, also gives **`pagination_class`**, **`filter_backend`**, etc

GenericViewSet: There are many GenericViewSet, the most common being **`ModelViewSet`**. They inherit from `GenericAPIView` and have a full implementation of all of the **actions**: `list`, `retrieve`, `destroy`, `updated`, etc. Of course, you can also pick some of them, read the docs
So, to answer your question: DRY, if you are doing something really simple, with a `ModelViewSet` should be enough, even redefining and calling `super` also is enough. For more complex cases, you can go for lower level classes.

Mixin

(The owner of the ice cream shop offered a basic flavor of ice cream (vanilla, chocolate, etc.) and blended in a combination of extra items (nuts, cookies, fudge, etc.) and called the item a "mix-in", his own trademarked term at the time.)

1, It provides a mechanism for **multiple inheritance** by allowing multiple classes to use the **common functionality**, **but without the complex semantics of multiple inheritance**.

2, **Code reusability**: Mixins are useful when a programmer wants to **share functionality between different classes**. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then included into each class that requires it.

3, Mixins allow inheritance and use of **only the desired features** from the parent class, not necessarily all of the features from the parent class.

Class-Based Views vs. Function-Based Views

Commonly, the **function-based views** are the most used due to them being the first used when **Django** views are starting to be understood and this view type is very easy to use and functional; so then, why were **class-based** views created? Which view type would be the most appropriate? What's the main **difference** between these two types of views?

Function-based views

“A view (function) is simply a **Python** function that takes a **Web request** and returns a **Web response**. This response can be the **HTML** contents of a **Web page**, or a **redirect**, or a **404 error**, or an **XML** document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response.“

*A simple **example** of a list function would be:*

```
def item_list(request):
        template_name = 'app/item_list.html'
        items = Item.objects.all()
        data = {}
        data['object_list'] = items
        return render(request, template_name, data)
```

As you can see, this **function** is **very easy** to implement and it’s very **useful** but the **main disadvantage** is that on a large **Django** project, there are usually a lot of similar functions in the views; one case could be that all objects of a **Django** project usually have **CRUD** operations, so this **code** is repeated again and again unnecessarily, and so, this was one of the reasons that the **class-based views** and generic views were created!

Class-based views

“**Class-based views** provide an **alternative way** to implement views as **Python** objects instead of functions. They do not replace **function-based views**, but have certain differences and **advantages** when compared to function-based views:

- Organization of code related to specific **HTTP methods** (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (**multiple inheritance**) can be used to factor code into reusable components.”

As already mentioned, the **class-based views** don’t replace **function-based views** but thanks to the inheritance they are easier to implement and more optimal, furthermore, for solved much more the repeated code, the **Django’s generic views** were developed and with this, the **class-based generic views** are **more optimal** yet.

*The **example** shown above would be:*

```
class ItemList(ListView):
        model = Item
```

This is very easy, isn't it?

The **generic views** help to **simplify** the **code** much more, in that it has **attributes** and **methods** providing functionality by default, if we wanted to modify or add attribute values or some method, we only have to write the **attribute/method** in our code and this will overwrite the default values.

Some generic views are: **TemplateView**, **ListView**, **DetailView**, **CreateView**, **DeleteView**, so on. See more here.

Conclusion

In conclusion, could it be said that it’s more appropriate to use **the class-based views** than the **function-based views**? Actually no, it’s just as good to use functions like use classes on the views, but it’ll depend on the functionality, we can use functions if the functionality is simple (show a template, for example), and we use generic views with class if the functionality contains **CRUD** operations or it’s more complex, as it's more optimal.

@ Django template

{% %} and **{{ }}** are part of **Django templating** language. They are used to pass the variables from **views** to **template**.

{% %} is basically used when you have an expression and are called **tags** while **{{ }}** is used to simply access the **variable**.

- What you have to do When you modify models.py.

```
C:\eclipse-workspace-msx\msxviewer> python manage.py makemigrations <projectname>
C:\eclipse-workspace-msx\msxviewer> python manage.py migrate
```

: **Before** you can get started with creating your own **models** and **views**, you must.

-Open **settings.py** file, **add** INSTALLED_APPS = [...., applicationname]

-**Create** a **Views**, Mapping **URLs**

@ How to log source file name & line number in Python

```
#!/usr/bin/env python
import logging

logging.basicConfig(format='%(asctime)s,%(msecs)d %(levelname)-8s
[%(filename)s:%(lineno)d] %(message)s',
    datefmt='%d-%m-%Y:%H:%M:%S',
    level=logging.DEBUG)

logger = logging.getLogger('stackoverflow_rocks')
logger.debug("This is a debug log")
logger.info("This is an info log")
logger.critical("This is critical")
logger.error("An error occurred")
```

Generates this output:

```
06-06-2017:17:07:02,158 DEBUG    [log.py:11] This is a debug log
```

```
06-06-2017:17:07:02,158 INFO     [log.py:12] This is an info log
06-06-2017:17:07:02,158 CRITICAL [log.py:13] This is critical
06-06-2017:17:07:02,158 ERROR    [log.py:14] An error occurred
```

@ csrf(cross site request forgery)

```
from django.middleware import csrf

def get_or_create_csrf_token(request):
   token = request.META.get('CSRF_COOKIE', None)
   if token is None:
      token = csrf._get_new_csrf_key()
      request.META['CSRF_COOKIE'] = token

   request.META['CSRF_COOKIE_USED'] = True
   return token
```

you should not use internal APIs and in fact _get_new_csrf_key() does not exist in Django anymore.
you can use get_token()

@ sqlite3

```
sqlite> .open "C:\\Devsup\\Data\\PRD.sqlite3"
sqlite> .database
main: C:\Devsup\Data\PRD.sqlite3
sqlite> .tables
```

To get a values_list from Model(Brand).

```
     brand_names = list(Brand.objects.values_list('name',  flat=True))
```

@ Django: using more than one database with inspectdb?

```
DATABASES = {
    'default': {DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    },

    'user_db': {
        'NAME': 'mydic',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'root',
        'PASSWORD': 'nbuser',
        'HOST': 'localhost',
        'PORT': '3306',
    },

    'mxhcv_db': {
        'NAME': 'mqxcv',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'root',
        'PASSWORD': 'nbuser',
        'HOST': 'localhost',
        'PORT': '3306',
    },
}
```

```
DATABASE_ROUTERS = ['myDjango_project.routers.MyApp2Router','path.to.AuthRouter',]
```

The order in which routers are processed is significant.
Routers will be queried in the order they are listed in the `DATABASE_ROUTERS` setting.

```
# Routers.py #
class MyApp2Router(object):
    """
    A router to control all database operations on models in
    the other applications.
    """

    def db_for_read(self, model, **hints):
        """
        Point all operations on myDjangoApp2 models to 'user_db'
                            myDjangoApp3 models to 'mxhcv_db'
        """
        if model._meta.app_label == 'myDjangoApp2':
            return 'user_db'
        elif model._meta.app_label == 'myDjangoApp3':
            return 'mxhcv_db'
        #return None
        # I recommend returning 'default' here since
        # it is your default database, this will allow
        # commonly used django apps to create their
        # models in the default database (like contenttypes and django auth
        return 'default'

    def db_for_write(self, model, **hints):
        """
        Point all operations on myapp models to 'other'
        """
        if model._meta.app_label == 'myDjangoApp2':
            return 'user_db'
        elif model._meta.app_label == 'myDjangoApp3':
            return 'mxhcv_db'
        #return None
        return 'default'

    def allow_syncdb(self, db, model):
        """
        Make sure the 'myDjangoApp2' app only appears on the 'other' db
        """
        if db == 'user_db':
            return model._meta.app_label == 'myDjangoApp2'
        elif model._meta.app_label == 'myDjangoApp2' or model._meta.app_label == 'myDjangoApp3':
            return False
        return None
```

You can inspect your second database with:
It creates models automatically by inspecting your db tables. it stored in our app files as ***models.py****. so we don't need to type all column manually.But read the documentation carefully before creating the models because it may affect the DB data.*

```
python manage.py inspectdb --database 'user_db'
```

You cannot inspect both at the same time.
You can specify a specific database like this:

*python manage.py **inspectdb --database=user_db > you_app_name**/models.py*

//**Dynamic db router**//
pip install django-dynamic-db-router and add DATABASE_ROUTERS=['dynamic_db_router.DynamicDbRouter']to your Django settings.

Manually selecting a database for a QuerySet
You can select the database for a **QuerySet** at any point in the **QuerySet** "chain." Just call **using()** on the **QuerySet** to get another **QuerySet** that uses the specified database.
using() takes a single argument: the alias of the database on which you want to run the query.
For example:

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()      or    >>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

requirements.txt for tangowithdjango

```
Django (1.11.2)
django-autoslug (1.9.3)
django-autoslug-field (0.2.3)
django-registration-redux (2.2)
mysqlclient (1.3.10)
```

Tails coding test

About this task

Think of this as an open source project. How would this have to look like, in order for you to be impressed with it - if you were to find it on Github? Now go do that.

Try to limit the amount of time you spend on this to 90-120 minutes. However, feel free to spend more - just make sure you're happy with your submission!

Hint: we're looking for a high-quality submission with great application architecture, not a "just get it done"-like approach.

What to do

If you're applying for a backend role

- Create a new Python-based application (any framework is fine, we prefer Flask)
- Render the list of stores from the `stores.json` file in alphabetical order through a backend template
- Use postcodes.io to get the latitude and longitude for each postcode and render them next to each store location in the template
- Build the functionality that allows you to return a list of stores in any given radius of any given postcode in the UK ordered from north to south and unit test it - no need to render anything

If you're applying for a full stack role

- Create a new backend application (any language is fine), using your favourite front-end framework (we'd prefer React, Vue or Ember though) on the user-facing side
- Build an API that returns stores from the `stores.json` file, based on a given search string in a performant way and unit test it, i.e. return "Newhaven" when searching for "hav" - make sure the search allows to use both city name and postcode
- Order the results by matching postcode first and then matching city names, i.e. "br" would have "Orpington" as the 1st result, "Bracknell" as the 2nd
- Build the frontend that renders the text field for the query and the list of stores that match it
- Add suggestions to the query field as you type, with a debounce effect of 100ms and a minimum of 2 characters

- Limit the results to 3 and lazy load the rest

Finally

- Include and render your favourite gif at the top right-hand corner of your application
- Zip your code up and upload it into Greenhouse
- Tell us what test you completed (backend or full-stack)
- Tell us what you'd have changed if you'd have had more time?
- What bits did you find the toughest? What bit are you most proud of? In both cases, why?
- What's one thing we could do to improve this test?