```
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 defect, 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

## @ How to create a Django project on Eclipse.

< **File - New - PyDev Django Project** >

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

1. cd to the directory where requirements.txt is located.
2. activate your virtualenv.
3. run: **pip install -r requirements.txt**

D:\venv35>Scripts\activate
(venv35) D:\venv35>**python -m pip install --upgrade pip**
(venv35) D:\venv35>**pip list**
(venv35) D:\venv35>**pip install django**
(venv35) D:\venv35>**pip install djangorestfraimework**
(venv35) D:\venv35>**django-admin startproject api_example**
(venv35) D:\venv35>**cd api_example**
(venv35) D:\venv35\api_example>**python manage.py migrate**

(venv35) D:\venv35\api_example>**python manage.py createsuperuser**
Username (leave blank to use 'tigerkt'):
Email address: kyungtak.lee@gmail.com
pw: amadeus88

(venv35) D:\venv35\api_example>**python manage.py startapp languages**

- **settings** : add apps (rest_framework, languages)
- **api_example/urls** :
- **languages/models** : - languages/urls :

(venv35) D:\venv35\api_example>**python manage.py makemigrations**
(venv35) D:\venv35\api_example>**python manage.py migrate**
(venv35) D:\venv35\api_example>**python manage.py runserver**

- **languages/serializers** :
- **languages/views** :

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

---