22. How to task in Django?

—> Django Background Task is an amazing task-scheduling library that lets developers automate and execute the background operations of a Django application. This helps the developers increase application speed, offload time-consuming activities and ensure a smoother user experience as well as run tasks at the scheduled time.

—> A Django app to run new background tasks from either admin or cron and inspect task history from admin.

—> To implement asynchronous tasks in Django, you can use Celery or :

How to use Django background task :

1. You would need to install Django Background Task using : pip install django-background-tasks

2. Next, add the installed app : INSTALLED_APPS = [
‘background-task’,
]

3. Create a new file called tasks.py

4. In the tasks.py file, define the task’s function.

```
from background_task import background
@background(schedule=60)
def my_scheduled_task():
    print(“Executing Scheduled Task Now...”)
```

5. Next, register the task function to be executed in the Django background task scheduler. You can all my_scheduled_task function in a view function or in method of your app’s AppConfig class.

```
from .tasks import my_scheduled_task
def register_task():
    my_scheduled_task.schedule(repeat=Task.DAILY, time=date.time(hour = 8
minute = 0))
```

6. Lastly, run this code on the terminal to start the background task :

```
python manage.py process_tasks
```

Features :

- Create async tasks either programmatically or from the admin
- Monitor async tasks from the admin
- log all tasks in the database for later inspection
- Optionally save task-specific logs in a TextField and/or in a FileField

pip install django-task

23. How to add a task in celery?

—> Follow the Steps to add tasks in Celery :

1. Install Celery : **pip install 'celery[redis]'**
2. Create a Django Project : **python -m django startproject (name)**
3. Configure Celery in Django : **Create celery.py**

```
from __future__ import absoulte_import, unicode_literals
import os
From celery import Celery
from django.conf import settings
os.environ.setdefault('DJANGO_SETTINGS_MODULE','your_pro.settings')
app = Celery('your_pro')
app.config_from_object('django.conf:settings', namespace = 'CELERY')
app.autodiscover_tasks(lambda:settings.INSTALLED_APPS)
```

4. Configure Celery Broker (Redis) : **settings.py**

```
CELERY_BROKER_URL = 'redis://localhost:6379/0''
```

5. Create Celery Tasks : **tasks.py**

```
from celery import shared_task
@shared_task
 def my_task(arg1, arg2):
      result = arg1 + arg2
      return result
```

6. Use Celery Tasks : You can now use celery tasks in your Django views, models or any other parts of your app. To call a task import it

```
from .tasks import my_task
Result = my_task.delay(3, 5)
```

7. Start Celery Worker : To run celery and process tasks you need to start a Celery worker process.

```
Celery -A your_pro worker --loglevel=info
```

8. **redis-server**
9. **redis-cli ping** : if it returns PONG then you're ready to go

24. Middleware :

—> In Django, middleware is a small plugin that runs in the background while the request and response are being processed. The application's middleware is utilized to complete a task.
—> Django Middleware is an essential component of the Django web framework, functioning as a lightweight, low-level plugin system. It seamlessly integrates into the request/response cycle, allowing for global modifications of incoming and outgoing requests. The flexible framework supports a range of operations like security, authentication, session management and CSRF protection. Middleware components, implemented as Python classes are defined in the MIDDLEWARE list in the project's settings.py file. Django not only offers built-in middleware but also enables developers to craft custom middleware solutions, enhancing the framework's adaptability and functionality.

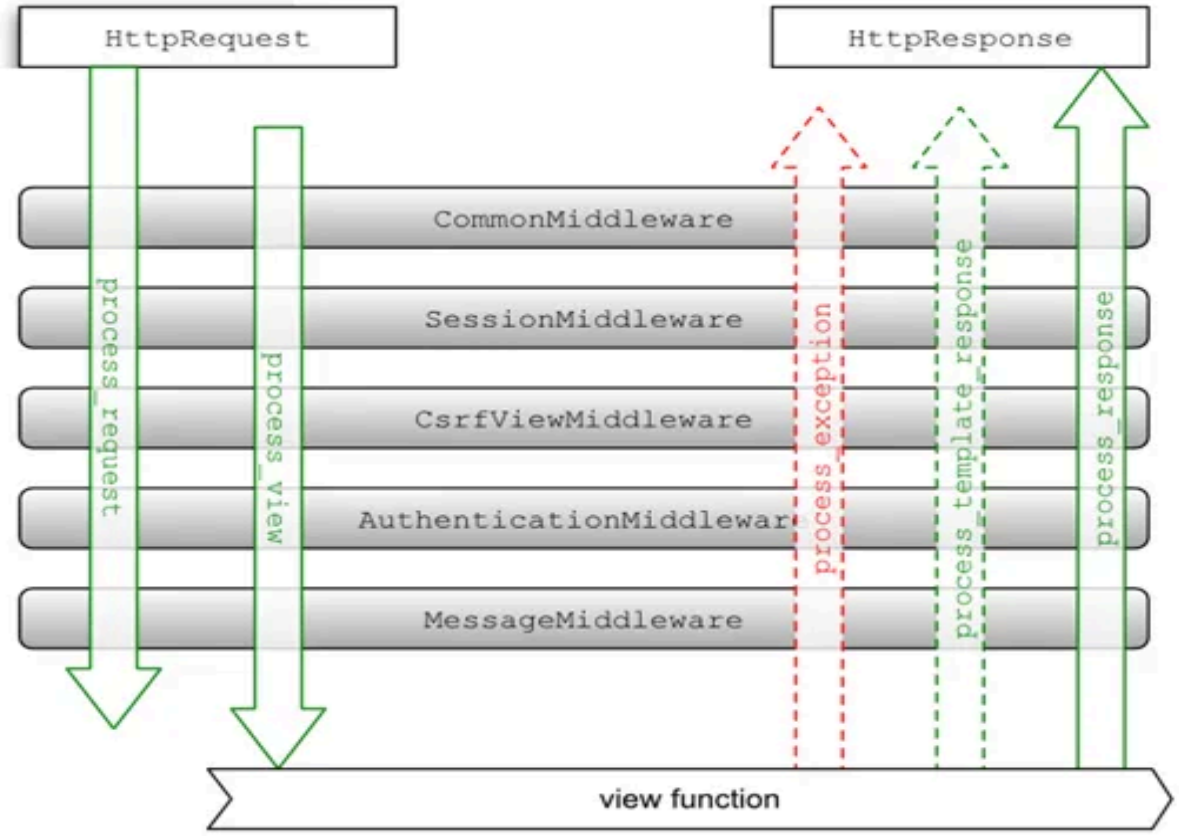
How does Middleware works :

—> The order in which we define the middleware classes is extremely important. The Middleware classes are processed sequentially that is from top-down during the request cycle :
1. SecurityMiddleware
2. SessionMiddleware
3. XFrameOptionsMiddleware

—> When the response cycle the Middleware classes are processed bottom-up :
1. XFrameOptionsMiddleware
2. MessageMiddleware
3. SecurityMiddleware

—> When a user makes a request from your application, a WSGI handler is instantiated, which handles the following things :

- Imports project's settings.py file and Django exception classes.
- Loads all the middleware classes which are written in the MIDDLEWARE tuple located in the settings.py file.
- Builds a list of methods which handle the processing of requests, views, responses & exceptions.
- Loops through the request methods in order.
- Resolves the requested URL
- Loops through each of the view processing methods.
- Calls the view function.
- Processes exception method (if any)
- Loops through each of the response methods in the reverse order from request middleware.
- Builds a return value and makes a call to the callback function.



25. Types of Middlewares :

—> There are two types of Middleware in Django :

1. Built-in Middleware
2. Custom Middleware

1. Built-in Middleware : They are provided by default in Dajngo when you create your project.

—> These are the default middlewares that come with Django :

- Cache Middleware
- Common Middleware
- GZip Middleware
- Message Middleware
- Security Middleware
- Session Middleware

- Site Middleware
- Authentication Middleware
- CSRF Protection Middleware

Authentication Middleware : It adds the user attribute, representing the logged-in user, to every incoming request object. If the user is not logged in then it will be set as an AnonymousUser Object.

Session Middleware : It helps you to store arbitrary data on a per-user basis like username, email, etc on the server side. On every request and response, a session_id is attached to the cookies that you can check in your browser through the console. Every time a request is made, session middleware gets the stored session data for the particular session_id and attaches it to the request object.

Message Middleware : It is used to display some notification message aka flash message to the user after form submission. You can store the message to the request object in your view and then show the message on your front end.

CSRF Middleware : It prevents Cross-Site Request Forgery attacks by adding hidden fields like CSRF_TOKEN to the POST forms and later validating for the correct value.

26. Custom Middleware :
2. Custom Middleware : You can write your own middleware which can be used throughout your project.
—> Create a Python Package(a folder with __init__.py inside) named as middleware.
—> Create a file named custom_middleware.py (or anything which you like) and a regular Python function/class in it.

—> You can write middleware as a function or a class whose instances are callable.

Function Based Middleware :

```
def simple_middleware(get_response):
    # One-time configuration and initialisation.
    def middleware(request):
    # Code to be executed for each request before the view (and later middleware) is called.
        response = get_response(request)
    # Code to be executed for each request/response after the view is called
        return response
    return middleware
```

Class-Based Middleware :

```
class ExampleMiddleware:
    def __init__(self, request):
        self.get_response = get_response
    def __call__(self, request):
        # Code that is executed in each request before the view is called
        response = self.get_response(request)
        # Code that is executed in each request after the view is called
        return response
    def process_view(request, view_func, view_args, view_kwargs):
        # This code is executed if an exception is raised
    def process_template_response(request, response):
        # This code is executed if the response contains a render() method
        return response
```

—> Now the final step will be to add your custom middleware in the MIDDLEWARE List in the settings.py file.

```
MIDDLEWARE = [

'your_app.middleware_directory.custom_middleware_file.CustomMiddleware_class',
                    ]
```

27. GITHUB

—> Add git before the following commands :

clone Clone a repository into a new directory

init Create an empty Git repository or reinitialize an existing one work on the current change (see also: git help everyday)

add Add file contents to the index

mv Move or rename a file, a directory, or a symlink

restore Restore working tree files

rm Remove files from the working tree and from the index

Examine the history and state (see also: git help revisions)

bisect Use binary search to find the commit that introduced a bug

diff Show changes between commits, commit and working tree, etc
grep Print lines matching a pattern
log Show commit logs
show Show various types of objects
status Show the working tree status

grow, mark and tweak your common history

branch List, create, or delete branches
commit Record changes to the repository
merge Join two or more development histories together
rebase Reapply commits on top of another base tip
reset Reset current HEAD to the specified state
switch Switch branches
tag Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)

fetch Download objects and refs from another repository
pull Fetch from and integrate with another repository or a local branch
push Update remote refs along with associated objects

28. ORM (Object Relational Mapping) :

—> The Django web framework includes a default object-relational mapping layer (ORM) that can be used to interact with data from various relational databases such as SQLite, PostgreSQL, and MySQL.

—> Django allows us to add, delete, modify and query objects, using an API called ORM.

—> ORM stands for Object Relational Mapping.

—> An object-relational mapper provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL Queries.

DJANGO ORM QUERIES :

—> Django lets us interact with its database models using an API called ORM. The Primary goal of ORM is to send data between a database and application models.

—> It represents a relationship between a database and a model.

—> The main advantage of using Django ORM Queries is that it speeds up and eliminates errors throughout the development process.

QuerySet :

—> A Query Set is a collection of data from a database. Query set allows us to get data easily, by allowing us to filter, create, order, etc. Lets, suppose we have a database table named Students.

—> Now, in views.py :

```
from django.http import HttpResponse
from django.template import loader
from .models import Members
def testing(request):
    mydata = Students.objects.all()
    template = loader.get_template('template.html')
    context = {
            'mystudents' : mydata,
            }
    return HttpResponse(template.render(context, request))
```

—> Template.html :

—> Check VS CODE

python manage.py shell

1. All objects :

Students.objects.all().values() : All Object's values will be printed

2. values_list() : It returns only column that is specified.

Students.objects.values_list('Subjects')

3. Create Objects :

from django.contrib.auth.models import Students

python manage.py shell

Students.objects.create(id=06, name="Victor", subject = "Ecology")

4. Filter Objects : The filter() returns a filtered search.
Students.objects.filter(name='Robert')

i. AND : We can get filtered data satisfying both of the queries mentioned.
Students.objects.filter(name='Robert',id=2).values()

ii. OR : We can also get filtered data that matches either of the queries mentioned.
Students.objects.filter(name='Jacob').values() | Students.objects.filter(name='Robert').values()

iii. Field Lookups : FieldLookups are keywords that represent specific SQL keywords.
Students.objects.filter(name__startswith='J').values()
—> There are some keywords for field lookups :
1. Contains : Contains the phrase
2. icontains : Contains the phrase but case-sensitive
3. endswith : Ends with
4. iendswith : Ends with but case-sensitive
5. startswith : Starts with
6. istartswith : Starts with but case-sensitive

5. Ordering Objects : Django provides a feature to sort the queries sets, using order_by().
1. To Sort the Result Alphabetically by Name :
Students.objects.all().order_by('name').values()
2. To Sort in descending Order :
Students.objects.all().order_by('-name').values()
3. Multiple Order by : To order more than one field
Students.objects.all().order_by('name', '-id').values()

6. Complex queries through method-chaining : A Query set can be combined with another query set by chaining them together,
Students.objects.filter(id=1).filter(subject='Chemistry')

Explanation of Each ORM Query and Associated SQL Queries :
—> Create a sample database using the django model
class Album(models.Model):
title = models.CharField(max_length=50)
artist = models.CharField(max_length=50)
genre = models.CharField(max_length=50)

```
        def __str__(self):
                return self.title
class Song(models.Model):
        name = models.CharField(max_length=50)
        album = models.ForeignKey(Album, on_delete=models.CASCADE)
        def __str__(self):
                return self.name
python manage.py makemigrations
python manage.py migrate

python manage.py shell
```

Get all records from a table(Model) :
Let us add some records for ease of explanation.

```
from playlist.models import Song, Album
>>>a = Album.objects.create(title="Add", artist="Sheeran", genre="pop")
>>>a.save()
>>>a = Album.objects.create(title="Abstract Road", artist="The Beatles",
genre="rock")
>>>a.save()
>>>a = Album.objects.create(title="Run Evolver", artist="The Beatles",
genre="slow")
>>>a.save()
>>>Album.objects.all()
```

Perform join operations in Django :
—> The SQL join unites data or rows from two or more tables that share a common field.

```
>>> a = Album.objects.create(title="Abstract Road", artist="The Beatles",
genre="rock")
>>> a.save()
```

```
>>> b = Song.objects.create(name = "Roadway", album=a)
>>> b.save()
>>> a = Album.objects.create(title="Test", artist="Sabss", genre="rock")
>>> a.save()
>>> b = Song.objects.create(name="Castle on the hill",album=a)
>>> b.save()
>>> q=Song.objects.select_related('album')
>>> q
```

Perform a truncate-like operation using Django ORM : To delete the records from the model

```
Album.objects.all().delete()
```

Conclusion :

- **The Django web framework includes a default object-relational mapping layer(ORM) that can be used to interact with data from various relational databases such as SQLite, PostgreSQL and MySQL.**
- **The main advantage of using Django ORM queries is that it speeds up and eliminates errors throughout the development process.**
- **Django allows us to add, delete, modify and query objects using an API called ORM.**
- **An object-relational mapper provides an object-oriented layer between relational databases and OOP languages without having to write SQL queries.**
- **The primary goal of ORM is to send the data between a database and application models.**
- **ORM represents a relationship between a database and a model.**
- **A Query Set is a collection of data from a database.**