**Django**

1) **Models :**

* Models in Django are Python classes that represent database tables.
* They define the structure of data stored in the database and provide an abstraction layer for interacting with the database.

provide an abstraction layer: means

Developers interact with models using high-level methods ,without write raw SQL queries.

* Django's ORM is database agnostic, meaning it supports multiple database backends such as PostgreSQL, MySQL, SQLite, Oracle, etc.
* Django takes care of translating the ORM queries to the appropriate SQL dialect for the configured database backend.

Models provide a **unified interface** for interacting with different database backends (e.g., SQLite, PostgreSQL, MySQL). Models **encapsulate business logic** and data access methods.

* + They define fields to store attributes like names, IDs, and dates.
  + Relationships between models are defined using Foreign Key, One To One Field, or Many To Many Field.

2) **Views :**

* In Django, views are Python functions or classes that **handle HTTP requests and return HTTP responses**.
* We define views to handle different types of **requests (CRUD),** such as **GET requests** for retrieving data, **POST requests** for submitting data, etc. Views interact with models to fetch or modify data.
* They encapsulate business (Rules and Processes,Data Manipulation and Transformation,Validation and Error Handling) logic for processing requests.
  + Views render templates or return JSON responses to the client.
  + They interact with models to fetch or modify data.
  + Different types of views handle various actions like displaying data, accepting form submissions, and processing user input.

**The choice of view type depends on your preference and the specific requirements of your project. Each view type offers different advantages, but they all can handle complex logic if implemented correctly.**

**Different types of Views:**

* **Function-Based Views (FBVs)**:

**Overview:** FBVs are simple functions that take a request object and return a response object. They are straightforward and flexible.

**Use Cases:**

* **Simple Logic:** When the view logic is straightforward and does not require additional methods or classes.
* **Rapid Prototyping:** For quick and simple implementations where the overhead of class-based views is unnecessary.
* **Small Applications:** Suitable for small applications
* **Flexibility:** All logic is in one function, making it easy to follow.

Example:

from django.http import HttpResponse

def hello\_world(request):

    return HttpResponse("Hello, World!")

* **Class-Based Views (CBVs)**:

 **Modularity:** Allows breaking down complex logic into methods, improving readability and maintainability.

 **Reusability:** Promotes code reuse through inheritance and mixins.

**Code Reusability:** When you need to reuse common behavior across multiple views.

Example using a generic view:

from django.views.generic import ListView

from .models import Article

class ArticleListView(ListView):

    model = Article

    template\_name = 'article\_list.html'

* **Generic Views**:

**Overview:** Generic views are pre-built views provided by Django for common tasks like displaying a list of objects, handling forms, and displaying detail views.

Types of Generic Views:

ListView:

Displays a list of objects retrieved from a model.

Automatically generates a list view based on a queryset and renders a template with the list of objects.

DetailView:

Displays details of a single object retrieved from a model.

Create, Read, Update, Delete view

* Includes generic views for **displaying lists of objects, detail views, form views**, and more.

Example using a generic ListView:

from django.views.generic import ListView

from .models import Book

class BookListView(ListView):

    model = Book

    template\_name = 'book\_list.html'

* **Template Views**:
* Views that render a Django template and return an HttpResponse.
* Useful for rendering HTML templates with dynamic content.

Example:

from django.shortcuts import render

def hello\_template(request):

    context = {'message': 'Hello, World!'}

    return render(request, 'hello\_template.html', context)

* **API Views**:
* Views designed specifically for building APIs using Django REST framework.
* Provide support for handling JSON responses, authentication, permissions, and more.

Example using DRF's APIView:

from rest\_framework.views import APIView

from rest\_framework.response import Response

class HelloWorldAPIView(APIView):

    def get(self, request, format=None):

        return Response({"message": "Hello, World!"})

* **Class-Based API Views**:
* Provide mixins and decorators for adding common API functionalities.

Example using DRF's APIView and ResponseMixin:

from rest\_framework.views import APIView

from rest\_framework.mixins import ListModelMixin

from rest\_framework.response import Response

from .models import Book

from .serializers import BookSerializer

class BookListAPIView(APIView, ListModelMixin):

    queryset = Book.objects.all()

    serializer\_class = BookSerializer

    def get(self, request, \*args, \*\*kwargs):

        return self.list(request, \*args, \*\*kwargs)

* **Template:** The template is a presentation layer. It web defines the structure of file layout to represent data in a page. It is an HTML file mixed with Django Template Language (DTL).

|  |
| --- |
|  |

3) **Serializers :**

* Serializers in Django REST Framework convert complex data (**querysets and model instances)** types to native Python datatypes that can be **easily rendered into JSON, XML,** or other content types.
* They handle serialization and deserialization of data.
  + They specify fields to include in serialized output and handle data validation.
  + Serializers are used with views to serialize or deserialize data when handling API requests.
  + Once the serialized data reaches its destination, deserialization is performed to reconstruct the original data structure from the serialized format.

**What is pickling and unpickling?**

- serialization while data transforming in that case pickle module comes into play

It is used to save and load Python objects to and from files, databases, or even across a network.

- Pickling:(serializes) object -> byte stream

- dumped as a file in the memory

- pickle.dump()

- Unpickling:(deserializes) byte stream -> objects

- loads the object to memory.

- pickle.load()

By understanding serialization and the role of the pickle module, you can efficiently manage data transformation processes, ensuring that data can be preserved, transmitted, and reconstructed as needed across different parts of your applications or systems.

4) **URLs :**

* URLs in Django are defined using URL patterns that map URL paths to views or other URL patterns.
* Two-tiered URL structure: project-level URLs and app-level URLs
* Project-level URLs route requests to specific apps, while app-level URLs define URL patterns specific to each app.
  + Project-level URLs serve as the entry point for incoming requests and route them to the corresponding app-level URLs for further processing.
  + Project-level URLs include app-level URLs using the **include()** function,
  + App-level URLs can define a URL namespace using the app\_name attribute
  + App-level URLs route incoming requests to the appropriate views within the app,
  + URLs facilitate navigation by providing a structured way to access different parts of the application.

**5)**  **Migrations :**

* Migrations in Django manage changes to the database schema over time.
* They allow creating, modifying, and deleting database tables and fields while keeping track of changes.
* We use the **makemigrations** command to generate migration files that contain the changes to be applied to the database schema..

**6)**  **Authentication and Authorization** :

* **Authentication** verifies **user identity**, while **authorization** determines **what actions users are allowed to perform within the system.**
* Django's built-in authentication system handles **login, logout, password management**, and **user registration**.
* **Role-based access control (RBAC)** assigns permissions based on user roles.
* 401 Unauthorized: The request requires user authentication.
* 403 Forbidden: The server understood the request but refuses to authorize it.
* [bi-de-n]-1+1+1=3=403

Authentication is the mechanism of incoming request with a set of identifying credentials. The [permission](https://www.django-rest-framework.org/api-guide/permissions/) and [throttling](https://www.django-rest-framework.org/api-guide/throttling/) policies can then use those credentials to determine if the request should be permitted.

REST framework provides several authentication schemes [ Basic, Session,Token, Remote Authentication] .and also allows custom schemes.

Authentication always runs at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

**Basic Authentication:**

Basic authentication is generally only appropriate for testing.

* **Overview**:
  + Basic authentication is a simple authentication scheme where the client sends its credentials (username and password) with every request.
  + Credentials are encoded in Base64 format and sent as an Authorization header in the HTTP request.
* **Workflow**:
  + Client sends a request to the server.
  + Server extracts the credentials from the Authorization header.
  + Server verifies the credentials against a user database or authentication provider.
  + If the credentials are valid, the server processes the request. Otherwise, it returns a 401 Unauthorized response.

## [**SessionAuthentication**](https://www.django-rest-framework.org/api-guide/authentication/#sessionauthentication)

Session authentication is appropriate for AJAX clients that are running in the same session context as your website.

If you're using an AJAX-style API with SessionAuthentication, you'll need to make sure you include a valid CSRF token for any "unsafe" HTTP method calls, such as PUT, PATCH, POST or DELETE requests.

**Session-based Authentication:**

* **Overview**:
  + Session-based authentication relies on the server creating and maintaining a session for each authenticated user.
  + After successful login, the server generates a unique session identifier (session ID) and stores it in a session store (e.g., database, cache).
* **Workflow**:
  + User logs in by providing credentials (username and password).
  + Server verifies the credentials and creates a new session for the user.
  + The session ID is returned to the client, typically as a cookie.
  + For subsequent requests, the client sends the session ID with each request.
  + The server validates the session ID and retrieves user information from the session store.

**Remote Authentication:**

* **Overview**:
  + Remote authentication involves delegating the authentication process to an external identity provider (IdP).
  + The client is redirected to the IdP's authentication endpoint, where the user authenticates.
  + Upon successful authentication, the IdP issues a token or assertion to the client, which is then used to access protected resources.
* **Workflow**:
  + Client requests access to a protected resource.
  + Server redirects the client to the IdP's authentication endpoint.
  + User authenticates with the IdP.
  + IdP issues a token or assertion to the client.
  + Client presents the token or assertion to the server to access the protected resource.

**Token-based Authentication:**

**JWT::**  A custom authentication system like JWT (JSON Web Tokens) **provides a stateless (meaning the server does not need to store session information)mechanism for authenticating and authorizing users in web applications**. JWT is a compact, self-contained token that encodes user information and claims, signed using a secret key or public/private key pair, allowing for secure transmission and verification of user identity.

* **Token Generation**: When a user logs in or authenticates, the server generates a JWT containing a payload with user information and claims, such as user ID, username, and roles. The token is signed using a secret key known only to the server.
* **Token Transmission**: The server sends the JWT to the client, typically in an HTTP header or as a cookie. The client stores the token locally, usually in the browser's localStorage or sessionStorage.
* **Token Verification**: the client includes the JWT in the request headers. The server verifies the token's signature, decodes the payload, and checks the claims to authenticate and authorize the user. If the token is valid and the user has the required permissions, the request is processed; otherwise, the request is denied.

SIMPLE\_JWT = {

    'ACCESS\_TOKEN\_LIFETIME': timedelta(minutes=5),

    'REFRESH\_\_TOKEN\_LIFETIME': timedelta(days=1),

    'ROTATE\_REFRESH\_TOKEN': True # if False then if we run the refresh token then it will not generate the refresh token only access token will generate .

}

JWTAuthentication

        Authentication\_classes = [JWTAuthentication]

        permission\_classes = [IsAuthenticated]

if we run it it throws the note 'Auth. credentials were not provided'

'''1 st generate the access token.  ['http POST http://127.0.0.1:8000/refreshtoken/refresh = 'token paste here...']  we shall get the access token by

http POST  [http POST http://127.0.0.1:8000/gettoken/username='user1' password='abc']'''

  access the JWT protected token :  http http://127.0.0.1:8000/studentapi/ 'Authorisation:Bearer paste the token'

  post some data in the DB :  http -f POST http://127.0.0.1:8000/studentapi/ name=Raj roll=103 city=Bokaro 'Authorisation:Bearer paste the token

## Permission:

## [**AllowAny**](https://www.django-rest-framework.org/api-guide/permissions/#allowany)

The AllowAny permission class will allow unrestricted access, **regardless of if the request was authenticated or unauthenticated**.

## [**IsAuthenticated**](https://www.django-rest-framework.org/api-guide/permissions/#isauthenticated)

The IsAuthenticated permission class will deny permission to any unauthenticated user, and allow permission otherwise.

This permission is suitable if you want your API to only be accessible to registered users.

## [**IsAdminUser**](https://www.django-rest-framework.org/api-guide/permissions/#isadminuser)

The IsAdminUser permission class will deny permission to any user, unless user.is\_staff is True in which case permission will be allowed.

[IsAuthenticatedOrReadOnly](https://www.django-rest-framework.org/api-guide/permissions/#isauthenticatedorreadonly)

The IsAuthenticatedOrReadOnly will allow authenticated users to perform any request. Requests for unauthenticated users will only be permitted if the request method is one of the "safe" methods; GET, HEAD or OPTIONS.

This permission is suitable if you want to your API to allow read permissions to anonymous users, and only allow write permissions to authenticated users.

## [**DjangoModelPermissions**](https://www.django-rest-framework.org/api-guide/permissions/#djangomodelpermissions)

it will provide only READ operation by default. bt if we want to access the POST UPDATE DELETE

permission with the same user then us user ko backend me ja kar wo sab model permission assign karna hoga.

exm = if POST req dena hey, dn ADD permission dena hoga particular user ko, SUPERUSER a login kore

avl user permission theke choosen user permission a particular permission transfer korte hbe

so this permission is more powerfull cz kake CREATE, UPDATE, DELETE korte debo seta amra thik korbo.  '''

## [**DjangoModelPermissionsOrAnonReadOnly**](https://www.django-rest-framework.org/api-guide/permissions/#djangomodelpermissionsoranonreadonly)

## [**DjangoObjectPermissions**](https://www.django-rest-framework.org/api-guide/permissions/#djangoobjectpermissions)

## [**Setting the authentication scheme**](https://www.django-rest-framework.org/api-guide/authentication/#setting-the-authentication-scheme)

The default authentication schemes may be set globally, using the DEFAULT\_AUTHENTICATION\_CLASSES setting. For example.

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.BasicAuthentication',

'rest\_framework.authentication.SessionAuthentication',

]

}

# Globally defined means applicable for each view classes

# global settings can be overriden by local

REST\_FRAMEWORK = {

    'DEFAULT\_AUTHENTICATION\_CLASSES' : ['rest\_framework.authentication.BasicAuthentication'],

    'DEFAULT\_PERMISSION\_CLASSES' : ['rest\_framework.permissions.IsAuthenticated']

**16. What are throttling classes in DRF, and how are they used?**

Throttling in DRF is used to limit the rate of requests that a client can make to an API. This helps to prevent abuse and ensures fair use of resources. DRF provides several built-in throttling classes like **AnonRateThrottle** and **UserRateThrottle**.

REST\_FRAMEWORK = {

    'DEFAULT\_THROTTLE\_CLASSES': [

        'rest\_framework.throttling.AnonRateThrottle',

        'rest\_framework.throttling.UserRateThrottle',

    ],

    'DEFAULT\_THROTTLE\_RATES': {

        'anon': '100/day',

        'user': '1000/day'

    }

}

Throttling protects your API from being overwhelmed by too many requests.

**7) Validation and error handling** :

* Validation ensures **data integrity and user input correctness**.
* Django's built-in form validation and model validation mechanisms to validate user input and ensure data integrity.
* Form validation🡪 **Client side**(HTML, JS) & **Server Side** validation framework.
* Model Validation 🡪 **DB level constraints**( required fields, data types, length limits), **Custom Validation Logic.**
* . For error handling, we use Django's built-in error handling mechanisms to catch and handle exceptions gracefully.

**8) Middleware :**

* Middleware in Django is a framework of hooks into Django’s request/response processing.
* . It allows you to modify request objects before they reach the view layer and response objects before they are returned to the client
* It's a lightweight, low-level plugin system for globally altering Django's input or output.
* **Use:-**Middleware can be **used for various purposes** like **authentication, session management, and error handling, CSRF protection.**
* **Custom middleware** can be written to add functionality such as logging, modifying request/response headers, **custom request/response processing**

**caching, compression, CORS handling** etc.

* Middleware is configured in the Django settings and executed in the order they are listed.

    'django.middleware.security.**SecurityMiddleware**',

    'django.contrib.sessions.middleware.**SessionMiddleware**',

    'django.middleware.common.**CommonMiddleware'**,

    'django.middleware.csrf.**CsrfViewMiddleware'**,

    'django.contrib.auth.middleware**.AuthenticationMiddleware'**,

    'django.contrib.messages.middleware.**MessageMiddleware',**

    'django.middleware.clickjacking.**XFrameOptionsMiddleware',**

 **Authentication and Authorization**:

* To enforce authentication or authorization policies.
* For instance, redirecting users to a login page if they are not authenticated.

 **Request and Response Modification**:

* To modify the request object before it reaches the view or to alter the response object before it is sent to the client.
* For example, adding custom headers or modifying the request data.

 **Performance Monitoring**:

* To monitor the performance of your application by measuring request processing time.
* For example, tracking how long each view takes to process requests.

 **Security**:

* To enforce security measures such as checking for specific headers, rate limiting, or blocking IPs.

**Step 1: Custom Middleware**

Create a new Python file named middleware.py inside your Django app directory (or you can add this code to an existing file).

# middleware.py

def my\_middleware(get\_response):

    print('one time initialisation')

    def my\_function(request):

        print('this is before view')

        response = get\_response(request)

        print('this is after view')

        return response

    return my\_function

Summary:

The custom middleware logs the request method and path when a request is received, and logs the response status code after processing the request. It provides a way to add custom processing or logging around each request and response in a Django application without modifying the individual views.

**9) ORM (Object-Relational Mapping):**

* ORM is a **programming technique** that allows developers to interact with a relational database using object-oriented programming **(OOPs) concepts**.
* Django's ORM is a **layer of abstraction** that allows developers to interact with databases using high-level Python objects rather than writing raw SQL queries.
* It provides a **mapping between database tables and classes** in the application, allowing developers to work with database records as objects.
* In Django's ORM, the class defined in a model represents a database table,by default the table name is 'appname\_class-name' . However, we can customize the table name using the Meta class within the model

The attributes (fields) of the class represent the columns of the table.

Each instance of the class corresponds to a row in the database table.

* + Django's ORM takes care of **translating** these Python objects into database tables and **performing CRUD** (Create, Read, Update, Delete) operations on them
  + **ORM handles tasks** such as database connection management, query execution, data retrieval, and result processing behind the scenes.

How Django processes the request and sends a response to the client:

1. **URL Routing**: Django's URL dispatcher routes the incoming request to the appropriate view based on the URL patterns defined in the **urls.py** file.
2. **Middleware Processing**: Django's middleware components process the request and response objects, performing tasks such as authentication, session management, CORS handling, and more.
3. **View Processing**: The view function or class-based view associated with the matched URL pattern is executed. The view performs the necessary logic, which may include querying the database, processing forms, or calling other functions and services.
4. **Model Interaction**: If the view interacts with a database through Django's ORM, the necessary database operations (e.g., querying, saving, updating) are performed to retrieve or manipulate data.
5. **Template Rendering** (for template views): If the view uses a Django template, the template engine renders the template with the context data provided by the view, generating the HTML content for the response.
6. **Response Creation**: A **HttpResponse** or **JsonResponse** object is created to encapsulate the response content generated by the view or template. The response object contains the HTTP status code, headers, and body content.
7. **Middleware Response Processing**: The response object is passed back through the middleware stack, allowing middleware to modify or decorate the response before it is returned to the client.
8. **Sending Response**: Finally, the processed response object is sent back to the client over the HTTP connection, completing the request-response cycle.

* **Why Is Django Called A Loosely Coupled Framework**

Django is often referred to as a "loosely coupled" framework **due to its design principles and architecture** that promote modularity, flexibility, and reusability of components.

1. Separation of Concerns:
   * Django encourages the separation of different components of a web application, such as models, views, templates, and URLs, into distinct layers or modules.
   * This separation allows developers to work on individual components independently without tightly coupling them together, making the codebase easier to maintain, test, and scale.

* **Informational Responses (1xx):**
* 100 Continue: The server has received the initial part of the request and the client should continue with the request.
* **Successful Responses (2xx):**
* 200 OK: The request was successful, and the server has returned the requested data.
* 201 Created: a new resource was created.
* 204 No Content: there is no content to send (often used for DELETE requests).
* **Redirection Messages (3xx):**
* 300 Multiple Choices: Indicates multiple options for the resource that the client may follow.
* 301 Moved Permanently: The requested URL has been permanently moved to a new location.
* 302 Found: The requested URL has been temporarily moved to a different location.
* 304 Not Modified: The client's cached version of the requested resource is still valid, and the server has not returned any content.
* **Client Error Responses (4xx):**
* 400 Bad Request: The server could not understand the request due to invalid syntax or missing information.
* 401 Unauthorized: The request requires user authentication.
* 403 Forbidden: The server understood the request but refuses to authorize it.
* 404 Not Found: The requested resource could not be found on the server.
* 405 Method Not Allowed: The HTTP method used in the request is not allowed for the requested resource.
* 429 Too Many Requests: The user has sent too many requests in a given amount of time.
* **Server Error Responses (5xx):**
* 500 Internal Server Error: A generic error message indicating an unexpected condition encountered by the server.
* 501 Not Implemented: The server does not support the functionality required to fulfill the request.
* 502 Bad Gateway: The server, while acting as a gateway or proxy, received an invalid response from an upstream server.
* 503 Service Unavailable: The server is currently unable to handle the request due to temporary overloading or maintenance.
* 504 Gateway Timeout: The server, while acting as a gateway or proxy, did not receive a timely response from an upstream server.
* **What do you mean by the CSRF Token?**

CSRF stands for Cross-Site Request Forgery, which is a type of **web security**.

CSRF attack occurs when a malicious website tricks a user's web browser into making a request to a different website where the user is authenticated.

Imagine you're logged into your online banking account in one browser tab and you visit a malicious website in another tab. The malicious website could secretly send a request to your banking website using your authenticated session, causing your browser to transfer money, change your password, or perform other actions without you realizing it.

To mitigate CSRF attacks, Django uses CSRF tokens, which are unique, random tokens generated for each user session. These tokens are included in forms and requests, and the server verifies them before processing actions to ensure that the request is legitimate and coming from the authenticated user, not from a malicious website.

.

A CSRF token is a unique, random value that is generated by the server and included in HTML forms and AJAX requests within a Django application. The token acts as a secret key that verifies that the request originated from a legitimate user and not from a malicious attacker trying to forge a request.

Here's how CSRF tokens work in Django:

1. Token Generation:
   * When a user loads a form or accesses a page with CSRF protection enabled, Django generates a CSRF token for that user's session.
   * The CSRF token is a random string that is unique for each session and is securely stored server-side.
2. Token Inclusion:
   * The CSRF token is included in the HTML form as a hidden input field (<input type="hidden" name="csrfmiddlewaretoken" value="...">).
   * For AJAX requests, the CSRF token is typically included in the request headers.
3. Token Verification:
   * When the user submits the form or sends an AJAX request, the server verifies the CSRF token included in the request.
   * If the token is valid and matches the expected value for the user's session, the request is processed normally.
   * If the token is missing, invalid, or mismatched, the server rejects the request with a CSRF token error, preventing the request from being processed and protecting against CSRF attacks.

Here's a tabular comparison between REST and FastAPI:

| **Aspect** | **REST**(Representational State Transfer): | **FastAPI** |
| --- | --- | --- |
| **Architecture** | Architectural style for designing networked applications | Web framework for building APIs with Python |
| **Performance** | Depends on implementation and design choices | High-performance due to asynchronous design and type hints |
| **Request Handling** | Typically synchronous request-response model | Asynchronous request-response model |
| **Input Validation** | Manual validation of request data | Automatic serialization and validation using Python type hints |
| **Documentation** | Manual creation of API documentation | Automatic generation of interactive API documentation |
| **Flexibility** | Flexible, supports various data formats and protocols | Flexible, supports multiple data formats and serialization methods |
| **Ease of Use** | Requires understanding of REST principles and HTTP methods | Easy to use with intuitive syntax and built-in validation |
| **Use Cases** | Wide range of web services and APIs, suitable for distributed systems | Building APIs quickly, real-time applications, microservices |

**Synchronous Request-Response Model:**

In a synchronous model, the client sends a request to the server and waits for the server to process the request and send back a response before proceeding with any further actions. During this time, the client is typically blocked and cannot perform other tasks.

**Example:**

1. **Client sends a request**: Suppose a client sends a request to a web server to fetch some data.
2. **Server processes the request**: The server receives the request, performs the necessary computations, and retrieves the requested data from the database.
3. **Server sends the response**: Once the server has processed the request and obtained the data, it sends the response back to the client.
4. **Client receives the response**: The client receives the response and proceeds with further actions based on the received data.

Asynchronous Request-Response Model:

In an asynchronous model, the client sends a request to the server but does not wait for the server to respond immediately. Instead, the client continues to perform other tasks while waiting for the response. When the response is ready, the client receives it and handles it asynchronously, without blocking.

Example:

1. **Client sends a request:** A client sends a request to a web server to perform a time-consuming operation, such as fetching large amounts of data.

2. **Server processes the request asynchronously:** The server receives the request and starts processing it. Instead of blocking, the server may perform other tasks or delegate the request to another process or thread.

3. **Client continues with other tasks:** While waiting for the response, the client can continue performing other tasks or handling other requests.

4. **Server sends the response asynchronously:** Once the server has processed the request and obtained the data, it sends the response back to the client.

5. **Client receives and handles the response asynchronously:** The client receives the response and handles it asynchronously, without blocking its main execution thread.

FastAPI:

FastAPI is a modern, fast (high-performance), web framework for building. It provides automatic serialization and validation of request and response data using Python type annotations and the Pydantic library. FastAPI is built on top of Starlette for the web parts and Pydantic for the data parts, making it a highly performant and efficient framework for building web APIs.

Infosis Questions

What is Ci cd ?

CI/CD stands for Continuous Integration and Continuous Delivery/Continuous Deployment. It is a set of practices and methodologies used in software development to automate the process of building, testing, and deploying applications.

1. **Continuous Integration (CI):** Continuous Integration is the practice of frequently integrating code changes into a shared repository. Each integration triggers automated tests to ensure that the changes do not introduce errors or regressions. The main goal of CI is to detect and fix integration errors as quickly as possible. CI systems often include tools for automating the build process, running unit tests, and performing code quality checks.
2. **Continuous Delivery (CD):** Continuous Delivery extends the principles of CI by automating the deployment process. It ensures that the code changes, once successfully integrated and tested, can be deployed to production or staging environments with minimal manual intervention. Continuous Delivery pipelines automate the steps required to package, release, and deploy applications, allowing teams to deliver software updates quickly and reliably.
3. **Continuous Deployment (CD):** Continuous Deployment takes the automation one step further by automatically deploying every code change that passes through the CI/CD pipeline to production environments. With continuous deployment, there is no manual intervention required to release software updates. It allows teams to deliver new features, fixes, and improvements to end-users rapidly and continuously.

Overall, CI/CD practices help teams to deliver high-quality software more efficiently by automating repetitive tasks, reducing manual errors, and accelerating the feedback loop between development, testing, and deployment. It promotes collaboration, agility, and innovation in software development processes.

**Soumik Project in details clarification**

In Django REST Framework(DRF**), viewsets.ModelViewSet** is a class-based view that **provides default implementations for commonly used CRUD operations** (Create, Retrieve, Update, Delete) for a model.

**viewsets**: This is a module within Django REST Framework that provides various types of viewsets for building APIs.

**ModelViewSet**: This specific viewset is designed to work with Django models. It provides actions such as list, retrieve, create, update, and destroy out of the box.

**By inheriting from viewsets.ModelViewSet, your StatusViewSet class automatically gains the following functionalities:**

* **Default Queryset and Serializer**: It sets the **queryset** attribute to **Status.objects.all()** and the **serializer\_class** attribute to **StatusSerializer**, which are used by default for performing CRUD operations.

# Create your views here.

class StatusViewSet(viewsets.ModelViewSet):

    queryset = Status.objects.all()

    serializer\_class = StatusSerializer

let suppose I have a model 'Books', I want to retrieve all the books which are published after friday, Today is monday. so write the django query

from datetime import datetime, timedelta

from django.db.models import Q

# Calculate last Friday's date

today = datetime.today()

last\_friday = today - timedelta(days=(today.weekday() + 3) % 7)

# Retrieve books published after last Friday

books\_published\_after\_friday = Book.objects.filter(pub\_date\_\_gt=last\_friday)

The **Q** object in Django's ORM is used to construct complex database queries with logical operators like **AND**, **OR**, and **NOT**. It allows you to build more complex and dynamic query conditions beyond simple field lookups.

* **today.weekday():**
  + **This function returns the day of the week as an integer, where Monday is 0 and Sunday is 6.**
  + **For example, if today is Thursday, today.weekday() would return 3.**
* **(today.weekday() + 3) % 7:**
  + **This expression calculates the number of days to subtract to get to the previous Friday.**
  + **We add 3 to shift the start of the week from Monday to Friday (Friday becomes day 4).**
  + **Then we use % 7 to ensure the result is within the range of 0 to 6 (days of the week).**
  + **For example, if today is Thursday (weekday = 3), (3 + 3) % 7 would result in 6.**
* **timedelta(days=(today.weekday() + 3) % 7):**
  + **This creates a timedelta object representing the number of days to subtract.**
  + **For example, if today is Thursday, timedelta(days=6) represents 6 days.**
* **last\_friday = today - timedelta(days=(today.weekday() + 3) % 7):**
  + **This subtracts the calculated number of days (6 days) from today's date to get the date for the previous Friday.**
  + **So, if today is Thursday, April 28, 2022, last\_friday would be assigned the value datetime(2022, 4, 22, 0, 0) (which is the date for the previous Friday, April 22, 2022).**

**pub\_date:**

**This refers to the field in the Book model that represents the publication date of the book.**

**Pagination:**

Technique used in **web development to break down large sets of data** into smaller chunks or pages.

**used in websites and web applications** to improve user experience

Here's how pagination typically works:

1. **Division of Data**: The dataset, such as search results, list of items, or database records, is divided into smaller subsets called pages. Each page contains a fixed number of items.
2. **Navigation Controls**: Pagination typically includes navigation controls, such as "Previous" and "Next" buttons, as well as page numbers
3. **Displaying Data**: When a user requests a page, only the data for that specific page is retrieved and displayed. This prevents the need to load and display the entire dataset at once, which can be inefficient and slow, especially for large datasets.
4. **Optimization**: Pagination helps optimize performance by reducing the amount of data transferred between the server and the client.

Top of Form

**Types of error**

1. **SyntaxError:**
   * Example: Missing colon in a function definition or using reserved keywords as variable names.
2. **IndentationError:**
   * IndentationError occurs when there are incorrect indentation levels in the code, violating Python's indentation rules.
   * Example: inconsistent indentation within a block of code.
3. **NameError:**
   * NameError occurs when a variable or name is not found in the current scope.
   * Example: or misspelling a variable name.
4. **TypeError:**
   * TypeError occurs when an operation or function is applied to an object of an inappropriate type.
   * Example: Performing arithmetic operations on incompatible data types or calling a function with the wrong number or type of arguments.

**TypeError**: argument of type 'int' is not iterable

1. **ValueError:**
   * ValueError occurs when a function receives an argument of the correct type but with an invalid value.
   * Example: Converting a string to an integer where the string does not represent a valid integer.
2. **IndexError:**
   * IndexError occurs when trying to access an index that is out of range in a sequence (such as a list or tuple).
   * Example: Accessing an element at an index that does not exist in the sequence.
3. **KeyError:**
   * KeyError occurs when trying to access a key that does not exist in a dictionary.
   * Example: Accessing a non-existent key in a dictionary using square bracket notation.

Timedelta module:

The **timedelta** class in the **datetime** module represents a duration of time, rather than a specific moment in time. It allows you to perform arithmetic operations on **datetime** objects,

often used to calculate time differences, add or subtract time from **datetime** objects,

| **Aspect** | **SQL (Relational Databases)** | **NoSQL (Non-Relational Databases)** |
| --- | --- | --- |
| **Data Model** | Structured with tables, rows, and columns | Dynamic with flexible structures like documents, key-value pairs |
| **Schema** | Fixed schema with predefined structure | Flexible schema with dynamic structure |
| **Query Language** | SQL | Varies depending on database type (e.g., JSON-like queries) |
| **Data Integrity** | Strong (Enforces ACID properties) | Flexible (Offers eventual consistency) |
| **Scalability** | Vertical (Scaling by increasing the power of the server) | Horizontal (Scaling by adding more machines) |
| **Transactions** | Fully ACID-compliant transactions | Eventual consistency, supports transactions in some cases |
| **Examples** | MySQL, PostgreSQL, SQLite, Oracle, MS SQL Server | MongoDB, Cassandra, Couchbase, Redis, Neo4j |
| **Use Cases** | Structured data, complex queries, transactions | Unstructured or semi-structured data, high scalability, real-time |

**Static Files:**  
In Django, static files are files that do not change dynamically and are used to serve assets such as CSS, JavaScript, images, and other files that are not tied to a particular Django application or view. Static files are typically stored in a directory named **static** within each Django app or in a centralized location specified by the **STATICFILES\_DIRS** setting.

**What Are Django Exceptions?**

Exceptions are designed to provide meaningful information about what went wrong and help developers diagnose and handle errors effectively.

Here are some common Django exceptions:

1. **ValidationError**: when a model's validation fails, typically when validating form data.
2. **Http404**: when a requested resource cannot be found, (page not found).
3. **PermissionDenied**: when a user lacks permission to perform a specific action, such as accessing a restricted view.
4. **SuspiciousOperation**: when Django detects a suspicious or potentially malicious operation, such as an attempt to tamper data.
5. **MiddlewareNotUsed**: This exception is raised to indicate that a middleware class is not being used in the current Django setup.

**What Are Django Signals?**

Django signals are a mechanism for allowing decoupled applications to get notified when certain actions occur elsewhere in the application. They provide a way for different parts of a Django application to communicate with each other without being directly connected or dependent on each other.

from django.contrib.auth.signals import user\_logged\_in,user\_logged\_out,user\_login\_failed

from django.contrib.auth.models import User

from django.dispatch import receiver

from django.db.models.signals import pre\_init,pre\_save,pre\_delete,post\_init,post\_save,post\_delete,pre\_migrate,post\_migrate

from django.core.signals import request\_started, request\_finished

from django.db.backends.signals import connection\_created

@receiver(user\_logged\_in,sender = User)     #2nd method to connect

def login\_success(sender,reqest,user,\*\*kwargs):

    print('--------------------')

    print('sender:', sender)

    print('user password:' , user.password)

    ## req codeing

#user\_logged\_in.connect(login\_success, sender= User)    # 1st method to connect

# scenarios

''' after successfully user's login you may capture many things. like if I want to capture the ip of the model ,

how many time he login ,you can count, I can.'''

1. **Events and Handlers**: Signals define events. such as when a model instance is saved or deleted. Handlers are functions that are connected to these signals and are executed when the corresponding event occurs.
2. **Built-in Signals**: such as model instance creation, modification, and deletion. For example, the **pre\_save** and **post\_save** signals are emitted before and after a model instance is saved, respectively.

from django.contrib.auth.signals import user\_logged\_in,user\_logged\_out,user\_login\_failed

after successfully user's login you may capture many things. like if I want to capture the ip of the model , how many time he login ,you can count,

the people who tried to log in but not to do so, i can capture there ip, user name. even if i want to restrict the login no like 1 can try maximum 4 attempt ,beyond that the ip will block. I can do this here.

1. from django.db.models.signals import pre\_init,pre\_save,pre\_delete,post\_init,post\_save,post\_delete,pre\_migrate,post\_migrate
2. **Custom Signals**: These signals can be emitted at any point in the code where an event of interest occurs, and handlers can be connected to them to respond to these events.
3. **Signal Senders and Receivers**: Signals have senders, which are objects that emit the signal, and receivers, which are functions that handle the signal. Receivers are connected to signals using the **@receiver** decorator or the **connect()** method provided by Django's signals framework.
4. **Decoupled Communication**: Signals enable decoupled communication between different parts of a Django application, allowing components to interact without being tightly coupled or dependent on each other.

**Nit 10) How to Delete/Insert/Update An Object Using QuerySet In Django?**

Queries :

Student.objects.get(name='kunal').update(name='mrunal')

Student.objects.filter(name='kunal').update(name='mrunal')

Create the name "palash" in the model

Student.objects.create(name='palash')

Retrieve the name "palash"

Student.objects.get(name='palash')

Delete the name "palash"

Student.objects.delete(name='palash')

QuerySet To Insert/Add An Object:

**new\_user = User(name = "Nitin Mangotra", city="Gurgaon") new\_user.save()**

User = model name

**Nit 11) How Can You Combine Multiple QuerySets In A View?**

**1. Using chain() from itertools:**

If both querysets belong to the different model,

***There is an issue with this approach, you won't get a queryset, you’ll get a list containing instances.***

from itertools import chain

from myapp.models import Model1, Model2

def my\_view(request):

    queryset1 = Model1.objects.all()

    queryset2 = Model2.objects.all()

    combined\_queryset = chain(queryset1, queryset2)

    # Now you can iterate over combined\_queryset

    for obj in combined\_queryset:

        # Process each object

**2. Using Union Operator (|):**

If you have two QuerySets with the same structure, both querysets belong to the same model.

queryset1 = Model.objects.filter(...)

queryset2 = Model.objects.filter(...)

combined\_queryset = queryset1 | queryset2

**4. Manually Combining Results:**

You can manually combine QuerySet results by converting them to lists and concatenating them:

queryset1 = Model1.objects.all()

queryset2 = Model2.objects.all()

combined\_results = list(queryset1) + list(queryset2)

**Nit-cq-1) Explain Django Admin & Django Admin Interface.**

Django Admin Panel:

❏ is a kind of graphical user interface that is used for administrative tasks.

❏ Django comes with a fully customizable in-built admin interface.

* It allows developers and administrators to perform CRUD operations on database records without writing any custom views or forms.

❏ You get the quick setup of the admin panel to manage your data and to access it.

❏ To use a database table with the admin interface, we need to register the model in the admin.py file.

❏ The development process becomes faster and also it becomes easy for the developers to perform

**administrative activities.**

❏ The application Django admin is imported from the django.contrib package.

❏ This imported application is also expected to get control by the corresponding organization hence it

does not require an additional front end.

The Django admin interface provides a number of advanced features like:

❏ Authorization access

❏ Managing multiple models

❏ Content management system

**Nit-cq-3) What Are The Advantages And Disadvantages Of Using Django?**

* Django follows the DRY or the Don’t Repeat Yourself Principle which means, one concept or a piece of data should live in just one place

Django is a Batteries Included Framework (wide range of built-in functionalities such as authentication, admin interface, ORM, forms handling, and more, reducing the need for third-party libraries.)

❏ Django Offers Rapid-development

❏ Django is highly Scalable

can handle high traffic websites with ease.

**Horizontal Scalability**:

* Adding more servers to distribute the workload. distributing incoming requests across multiple servers.

**Vertical Scalability**:

* upgrading the hardware resources of the server, such as CPU, memory, and storage.

**Caching and Optimization**:

* Django provides built-in support for caching mechanisms, which can significantly improve application performance and scalability by reducing database load and response times.

**Asynchronous Processing:**

* can improve scalability by handling concurrent requests more efficiently
* Django supports asynchronous programming paradigms through libraries like **Django Channels** ,**Celery.**

❏ Django provide high Security

* Django provides template system to define HTML template for your web page

**Explain Q objects in Django ORM?**

**Q** is an object to encapsulate a collection of keyword arguments specified as FIELD LOOKUPS

**Scenario: Retrieve all active users who have made a purchase in the last month.**

from django.utils import timezone

from myapp.models import User, Purchase

active\_users = User.objects.filter(is\_active=True, purchase\_\_timestamp\_\_gte=timezone.now() - timezone.timedelta(days=30)).distinct()

**Timestamp:** Used to represent dates and times as a single number (Unix timestamp), facilitating easy manipulation and comparison of datetime data.

**Timezone:** conversions between different time zones, ensuring accurate handling of datetime data across geographic regions.

**Timedelta:** Enables the representation and manipulation of durations of time, useful for performing arithmetic operations on datetime objects, calculating dates in the past or future, and implementing timeouts or scheduling tasks based on time intervals.

**Scenario**: Retrieve all users who registered within the last week.

from django.utils import timezone

from myapp.models import User

users\_last\_week = User.objects.filter(date\_registered\_\_gte=timezone.now() - timezone.timedelta(days=7))

**date\_registered\_\_gte** is not a variable; it's a lookup expression used in Django QuerySet filtering.

Here's what each part of **date\_registered\_\_gte** represents:

* **date\_registered**: This is the field name in the Django model. It refers to a field named **date\_registered** in the **User** model, which presumably stores the date and time when a user registered.

1. **Why use Time Zones**: Django applications can be deployed in various regions, and users may be located in different time zones. By using **timezone**, we ensure that datetime operations are consistent regardless of the server's or user's time zone.

The lookup operator in Django for "less than or equal to" is **lte**, which stands for "less than or equal".