Explain the Model-View-Controller (MVC) architectural pattern in Django.

Django follows a design pattern called Model-View-Controller (MVC), which is sometimes referred to as Model-View-Template (MVT) in the context of Django. The MVC/MVT pattern is a software architectural pattern that separates an application into three interconnected components, each with distinct responsibilities

The Model represents the data structure and database schema. The View handles the presentation logic and rendering of data. The Template defines the HTML structure. This separation enhances code organization, making development and maintenance more manageable.

What is the purpose of Django apps?

Django apps are modular components that promote code reusability. Each app handles a specific functionality within a project. This modular structure improves maintainability and allows developers to plug in or reuse apps across different projects.

What is the Django admin site and how do you register a model with it?

The Django admin site is a built-in, powerful, and customizable administrative interface provided by the Django web framework. It's designed to make it easier for developers and administrators to manage and interact with the application's data without having to write custom administrative views and templates.

To register a model, you create an admin.py file within the app and use the admin.site.register(ModelName) method.

9.

What is a QuerySet in Django and how is it different from a raw SQL query?
Hide Answer

A QuerySet is a collection of database queries represented in a Pythonic way. It allows you to retrieve, filter, and manipulate data from the database using Python code, without writing raw SQL queries. This abstraction enhances code readability and maintainability.

QuerySets offer several advantages over raw SQL queries:

Pythonic and ORM-Based:

- QuerySets are Python objects, not raw SQL strings, which makes them more readable and maintainable.
- They use Django's Object-Relational Mapping (ORM) system, allowing you to work with database records as Python objects.

Database Agnostic:

- QuerySets are database-agnostic, meaning you can write queries that work with different database backends (e.g., PostgreSQL, MySQL, SQLite) without modification.

How can you retrieve data from the database using Django's ORM?

You can retrieve data using the Django ORM by using methods like .objects.all(), .filter(), .get(), and more on a model's QuerySet. These methods generate SQL queries and return Python objects, making database interactions more intuitive.

```
from django.db import models

# Create your models here.

class UserProfile(models.Model):

        first_name = models.CharField(max_length=50, null=True, blank=True)

        last_name = models.CharField(max_length=50, null=True)

        email = models.EmailField(null=False, blank=True)

        birthdate = models.DateField(null=False)
```

- **first_name** can be empty in both the database and forms because it has both null=True and blank=True.
- **last_name** can be empty in the database but is required in forms because it has null=True (for the database) but does not have blank=True (for forms).
- **email** cannot be empty in the database but is optional in forms because it has null=False (for the database) but has blank=True (for forms).
- **birthdate** cannot be empty in both the database and forms because it has neither null=True nor blank=True

• Optional Form Fields: Use blank=True when you want to allow users to leave certain form fields empty. For example, a blog post's introductory title might be optional, so you'd set blank=True for the title field.

• Custom Form Validation: If you plan to implement custom validation logic for a field, blank=True can be useful in cases where you want to allow empty values but still perform additional validation checks.

Django on_delete

The on_delete is one of the parameter which helps to perform database-related task efficiently. This parameter is used when a relationship is established in Django. The on_delete parameter allows us to work with the foreign key.

It is clear that whenever the foreign key concept comes into the scenario, the **on_delete** parameter is expected to be declared as one among the parameters in the foreign key.

On_delete = models.CASCASE

When we set the **on_delete** parameter as **CASCADE**, deleting the reference object will also delete the referred object. This option is most useful in many relationships. Suppose a post has comments; when the Post is deleted, all the comments on that Post will automatically delete. We don't want a comment saving in the database when the associated Post is deleted.

```
1. rom django.db import models
2. 
3. # Create your models here.
4. 
5. class Auther(models.Model):
6.     first_name = models.CharField(max_length=30)
7.     last_name = models.CharField(max_length=30)
8.     email = models.EmailField()
9. 
10.    def __str__(self):
11.        return "%s %s" % (self.first_name, self.last_name)
12. 
13. class Post(models.Model):
14.    title = models.CharField(max_length=100)
15.    # Here we define the on_delete as CASCADE
16.    author = models.ForeignKey(Auther, on_delete=models.CASCADE)
17. 
18.    def __str__(self):
19.        return self.title
```

We have created the two models, Author, and Post. In the post model, we define a foreign key field named **Author** referencing the Author's object. Then we define the on_delete parameter as **CASCADE.**