Devinterview-io / **django-interview-questions** Public

🟣 Django interview questions and answers to help you prepare for your next technical interview in 2024.

50 stars 11 forks Branches Tags Activity

Star Notifications

Code Issues Pull requests Actions Projects Security Insights

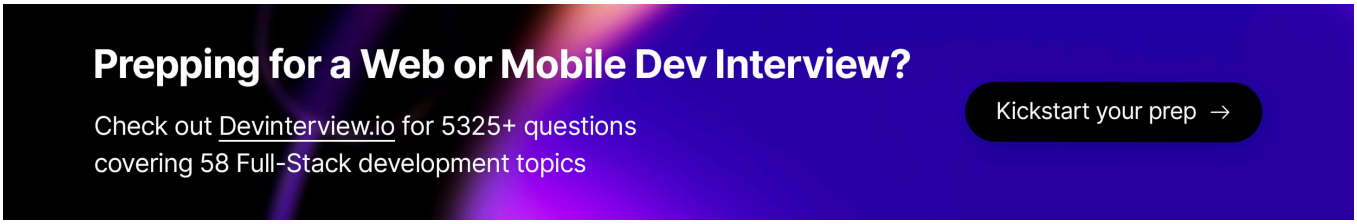
main 1 Branch 0 Tags Go to file Go to file Code

Devinterview-io Update README.md 65b79f7 · last year

README.md Update README.md last year

README

# Top 70 Django Interview Questions




**You can also find all 70 answers here 👉 Devinterview.io - Django**

## 1. What is *Django* and what are its key features?

**Django** is a high-level Python web framework celebrated for its emphasis on simplifying complex tasks and following the "Don't Repeat Yourself" (DRY) and "model-template-views" (MTV) paradigms. It's renowned for rapid development, robustness, and vast ecosystem of packages.

### Key Features

- **Object-Relational Mapping** : Enables database interactions using Python objects.
- **Admin Panel** : Generates a user-friendly interface for database management.
- **URL Mapping** : Routes web requests based on URLs, using `urls.py` files.
- **Template Engine** : Processes HTML templates, separating design from logic.
- **Form Handling** : Simplifies form validation and rendering.
- **Security** : Offers built-in protections against common web vulnerabilities.
- **Middleware** : Allows global request/response customization.
- **Simplified Queries** : Provides a high-level query API for database operations.
- **Shared Components** : Supports pluggable apps for easy component sharing.
- **Auto-Documentation** : Generates documentation for models and their attributes.
- **File Handling** : Provides utilities for file uploads and serving.
- **Asynchronous Support** : Enhanced in recent versions to handle asynchronous tasks.
- **Versatility** : Compatible with various web servers, databases, and front-end frameworks.
- **Scalability** : Adaptable to large, high-traffic projects.
- **Package Ecosystem** : A rich collection of "pypi" packages complements Django's core features.
- **Built-in Cache** : Offers caching support for performance optimization.
- **Internationalization** : Facilitates multi-language support.
- **REST Framework** : Offers extensive support for building RESTful APIs.

## 2. Explain the *MTV (Model-Template-View)* architecture pattern in *Django*.

**Django** is built around the MVT (Model-View-Template), which is nearly identical to the more commonly known **MVC** (Model-View-Controller) pattern. Here's a breakdown of the core components in Django's MVT:

### MVT Components

- **Model** (MVC equivalent): Responsible for data access and business logic. A model in Django is typically a Python class that represents a database table.
- **View** (MVC equivalent: Responsible for presenting data): Handles user input, processes requests, and returns appropriate responses. In newer versions of Django, the view is more akin to a controller and is responsible for the logical flow of the application.
- **Template** (MVC equivalent): Responsible for the presentation and user interface. A template in Django is an HTML file that utilizes its templating language to dynamically render data.

### Request-Response Lifecycle

1. **Client Request**: A user initiates an action, for example by clicking on a link in a web browser.
2. **URL Dispatcher**: The URL dispatcher (in newer versions, the 'path' or 're_path' function) maps the incoming URL to a corresponding view.
3. **View Processing**: The view performs any necessary logic, such as retrieving data from the database using models.
4. **Response Building**: The view sends the data to a template for rendering and then returns an HTTP response to the client.
5. **Template Rendering**: If a template is used, it processes the data and renders the HTML, which is then included in the response.

### Relationship with MVC

- **Model**: Adheres closely to the MVT and MVC paradigms, representing data and business rules.
- **View vs. Controller**: In MVT, the view correlates more closely to the traditional concept of a controller. This is because it processes incoming requests, interacts with models as needed, and oversees the flow of the application.
- **Template vs. View**: The MVT view is akin to the MVC view, responsible for presenting data to the user. The MVT view, however, also processes user requests, whereas the MVC view is more passive and simply displays data provided by the controller. The MVT template is analogous to the MVC view in that it focuses on the presentation layer.

## 3. What is a *Django project* and how is it different from a *Django app*?

**Django Project** is the high-level umbrella under which you build a web application. It encompasses multiple components such as settings, URLs, and can host several apps.

In contrast, a **Django App** is a standalone module designed to serve a specific functionality or business area, following the concept of "application" as a group of related features.

### Key Components of a Django Project

- **Settings**: Configuration options for the project and its apps.
- **URLs**: Defines the endpoint mappings using `urls.py`.
- **WSGI/ASGI**: Entry points for the web server to serve the project, handling HTTP and WebSocket requests, respectively.

### Key Components of a Django App

- **Models**: Data layer, defining data structures using models.py.
- **Views**: Business logic and presentation layer.
- **Templates** (optional): Presentation layer, housing HTML and rendering logic.
- **URLs**: Endpoint mappings for the app, known as "scoped URLs".

### Code Example: Project and App Structure

Here is the folder structure.

```
myproject/     # Django Project
    manage.py  # Project Management Utility
    myproject/ # Project Directory
```

```
        settings.py
        urls.py
        wsgi.py
        asgi.py
    myapp1/    # Django App 1
    myapp2/    # Django App 2
    ...
```

It shows the typical structure where a project contains one or more apps.

## 4. Describe the purpose of the *settings.py* file in a *Django project*.

The `settings.py` file in a Django project is crucial for configuring the project, including its applications and external resources. It allows for **customization** and **control** over various aspects of the Django project.

### Key Features

- **Global Settings**: Manages project-wide configurations such as installed applications, middleware, URLs, and more.
- **Dynamic Configuration**: Utilizes environment variables to secure sensitive data and allows for different configurations in development, testing, and production environments.
- **Security and Debugging**: Provides options for CSRF protection, session management, and detailed error handling.
- **Database Setup**: Offers flexibility to work with different databases, including SQLite, MySQL, PostgreSQL, and others.
- **Internationalization and Localization**: Facilitates multi-language support for web applications.
- **Customization and Extensibility**: Supports third-party app integration and allows for the creation of custom `AppConfig` classes.

### Importance of settings.py

- **Centralized Configuration**: Eliminates the need for scattered config files and ensures all settings are conveniently located.
- **Adaptability**: Its modifiability caters to evolving project requirements and changing deployment environments.
- **Consistency**: Promotes a uniform configuration setup across development, testing, and production stages.
- **Version Control**: In multi-developer setups, tracking changes to this file ensures everyone is on the same configuration page.

### Example Use-Cases

1. **Installed Applications**:
    - Identifying project-specific apps for features such as authentication, REST APIs, etc.
2. **Middleware**:
    - Global request/response handling, like CORS or authentication.
3. **Database Configuration**:
    - Choosing the target database, setting up primary database connections.
4. **Static and Media Files**:
    - Configuring file storage for user-uploaded content.
5. **Internationalization**:
    - Enabling and configuring multi-language support.
6. **Security and Debugging**:
    - Managing settings like `DEBUG` mode, CORS policies, and SSL/HTTPS requirements.
7. **Custom AppConfigs**:
    - Fine-tuning settings for specific Django apps or handling app initialization routines.
8. **Environmental Variables**:
    - Using keys, secrets, or sensitive data from the environment for security and flexibility.

9. **Testing and CI**:
    - Adjusting configurations for automated testing and continuous integration pipelines.

## 5. What is the role of the *urls.py* file in a *Django project*?

In Django, the `urls.py` file plays a key role in **routing** and **mapping URLs** to views. Each app typically has its own `urls.py` for modular URL handling.

### Global URLs vs. App URLs

- **Global (`project.urls`)**: The project's main `urls.py` generally includes paths to various apps.
- **Local (`app.urls`)**: Each app's `urls.py` handles its specific URL mappings.

### URL Patterns

Every `urls.py` file uses URL patterns, defined with `path()` or `re_path()`.

- **path()**: For simple text-based URLs.
- **re_path()**: For complex URLs using regular expressions.

### Passing URLs to Views

The `path()` function's `view` argument typically points to the corresponding view function. You can also send **additional data**, like query parameters or URL segments.

**Example: path()**

```
from django.urls import path
from . import views

urlpatterns = [
    path('articles/2023/', views.special_case_2003),
]
```

In the example, the URL pattern directs any request to `/articles/2023/` to the `special_case_2003` view.

**Example: path() with Parameters**

```
from django.urls import path
from . import views

urlpatterns = [
    path('articles/<int:year>/', views.year_archive),
]
```

Here, the URL pattern sends any matching URL (like `/articles/2022/`) and the extracted year to the `year_archive` view.

### URL Handling Best Practices

- **Consistency**: Follow a clear and standardized URL structure.
- **Namespacing**: Employ app namespaces to avoid URL name clashes.
- **Template Integration**: Use `reverse()` and `redirect()` to decouple URLs from views.

## 6. Explain the concept of *Django's ORM (Object-Relational Mapping)*.

**Django's ORM** bridges the gap between relational databases and Python objects. This integration allows you to perform database operations using high-level Pythonic methods, rather than direct SQL queries.

### Core Components

The three fundamental components of Django's ORM are:

1. **Models**: Defined in `models.py`, they specify the database structure in a class-based syntax.
2. **Model Instances**: These are objects created from your models and represent specific database records.
3. **QuerySets**: These are high-level, chainable, and lazy-evaluated database queries that allow you to interact with your model instances.

### Key Concepts

- **Model Classes**: These are Python classes that inherit from `django.db.models.Model`. Each class attribute represents a database field.
- **Field Types**: Django provides a range of field types (such as `CharField`, `IntegerField`, and `ForeignKey`) to cover diverse data requirements.
- **Model Relationships**: Models can be related to one another, for example, in a one-to-many or many-to-many configuration.
- **Manager Methods**: Objects of the `models.Manager` class provide utility functions for database access. Every model has at least one manager, typically named `objects`.
- **QuerySet Methods**: The `objects` attribute of a model provides a `QuerySet`. This can be filtered, sorted, and manipulated in various ways before evaluation.

### Benefits of Using ORM

- **Portability and Flexibility**: Applications developed on top of the ORM can quickly adapt to different database backends.
- **Security**: The ORM offers protections against common security threats like SQL injection.
- **Productivity**: Higher abstraction levels reduce the need for intricate database-specific code, resulting in quicker development cycles.

### Code Example: ORM in Action

Here is a simple model in Django:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    published_date = models.DateField()

    @property
    def is_recent(self):
        return self.published_date > some_logic_to_get_recent_date()

# Querying examples:

# Get all books published in the last year, written by authors with a name starting with 'A', and order them by title
recent_books = Book.objects.filter(published_date__gt=one_year_ago, author__name__istartswith='a').order_by('title')

# Get the top 5 authors with the most books:
top_authors = Author.objects.annotate(num_books=models.Count('book')).order_by('-num_books')[:5]
```

## 7. What is a *Django model* and how is it defined?

**Django Models** form the architectural foundation for database-driven applications. A model serves as a data structure and includes essential fields and behaviors.

### Model Definition

A model class is a Python **subclass** of `django.db.models.Model`. Each attribute within the class represents a database field.

The model class specifies the database table name, any data relationships, metadata, and methods for data manipulation.

### Key Model Components

#### Fields

- **Definition**: Represent the table's columns.
- **Types**: e.g., `CharField`, `DateField`.
- **Primary Key**: `id` is default or can be customized.
- **Descriptors**: Define field attributes e.g., `null`, `blank`.

#### Meta Options

- **table_name**: Database table name.
- **unique_together**: A list of tuples for fields that should be unique when considered together.
- **ordering**: Default result order.

#### Methods

- **Objects**: Custom data managers.
- **__str__()**: Human-readable instance name for admin and more.

#### Example: Model Code

Here is the Django code for the `Author` and `Book` model which demonstrates relationships between the two models:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    birth_date = models.DateField()

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    published_date = models.DateField()
    genres = models.ManyToManyField('Genre')

    def __str__(self):
        return self.title

class Genre(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name
```

In this example:

- `Author` and `Book` are model classes, each with its own table and fields in the database.
- `Author` has a `ManyToMany` relationship with `Genre` through the `Book` model.
- `ForeignKey`, linking `Book` to `Author`, establishes a one-to-many relationship.

The `ForeignKey` indicates each Book relates to one Author, while the cascade `on_delete` specifies that if an Author is deleted, all their books are also deleted.

## 8. Describe the purpose of *Django's admin interface*.

The Django admin interface is a powerful tool that automatically generates a user-friendly interface for performing common data management tasks, such as creating, reading, updating, and deleting (CRUD operations) for your application's models.

### Key Features

- **Quick Implementation**: Provides out-of-the-box tools for data management, requiring minimal setup.
- **Model-Centric Interface**: Operations are organized based on your application's data models.
- **DRY (Don't Repeat Yourself) Philosophy**: Changes to your models are automatically reflected in the admin interface.

### Common Admin Actions

- **Database Records**: View, add, edit, and delete records.
- **Data Relationships**: Navigate through foreign key and many-to-many relationships.
- **Data Validation**: Basic field-level validation is performed.

### When to Use and When Not to Use the Admin Interface

#### When to Use

- **Rapid Prototyping**: For quick feedback or proof of concept.
- **Administrative Tasks**: For internal tools or during early project stages.
- **Quick Data Fixes**: Ad-hoc data corrections or debugging.

### Code Example: Setting Up the Admin Interface

Here is the Python code:

```
from django.contrib import admin
from .models import MyModel

@admin.register(MyModel)
class MyModelAdmin(admin.ModelAdmin):
    list_display = ('field1', 'field2', 'some_method', 'related_field__nested_field')
    list_filter = ('field1', 'field2')
    search_fields = ('field1', 'field2', 'related_field__nested_field__name')

    def some_method(self, obj):
        return obj.field1 + " - " + obj.field2
    some_method.short_description = 'Custom Description'
```

### When Not to Use

- **Production Client-Facing UI**: The admin interface is not designed for public, customer-facing views. It's preferable to create custom views using Django's forms and templating for public consumption.
- **Complex Data Operations**: For intricate data management or workflows, a custom UI provides better control and user experience.

### Admin Interface in urls.py

Here is the Python code:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
    # ...
]
```

## 9. What is a *Django view* and how is it created?

A **Django view** functions as an endpoint for web requests. It retrieves data from a database, processes it as needed, and then returns a response, which can be an HTML page, a redirect, or a JSON object, among others.

### Types of Views

- **Function-Based Views** (FBV): These are created using functions.
- **Class-Based Views** (CBV): These are created using classes and offer a more structured approach, often with built-in features like HTTP method handling and view mixins.

### Steps to Create a Django View

1. **Define the View Function or Class**: This entails specifying the unique logic for the view. In the case of a function-based view, it's a Python function with the `@` decorator and an HTTP method. For a class-based view, you define a class with specific method names for different HTTP methods.

2. **Map the View to a URL**: Every view must be associated with a URL pattern to be accessible. This is orchestrated in the `urls.py` file. You can use either `path` or `re_path` for simple or advanced URL matching, respectively.

3. **Handle the Request and Generate a Response**: In the view, the HTTP request is received and parsed as needed. The response should adhere to the view's requirements, which could mean rendering a template, redirecting the user, or returning specific data types (e.g., JSON).

### Code Example: Function-Based View

Here is the Django view-specific code:

```
# views.py

from django.http import HttpResponse
from django.shortcuts import render

def my_view(request):
    # Business logic, such as database queries or form processing
    data = ...

    # Return a rendered template or a custom HTTP response
    return render(request, 'template.html', {'data': data})
```

### Code Example: Class-Based View

Here is the Django view-specific code:

```
# views.py

from django.views import View
from django.http import JsonResponse

class JSONResponseView(View):
    def get(self, request):
        return JsonResponse({'key': 'value'})

    def post(self, request):
        return JsonResponse({'key': request.POST['data']})
```

### Routing and URL configuration

In a Django **MVC** (Model-View-Template) setup, the **URL configuration** functions as a mediator between the view and the model, handling incoming HTTP requests and ensuring an appropriate view responds.

In more complex web applications, this strategy allows for a clear separation of concerns, making both maintenance and expansion more straightforward.

## 10. Explain the concept of *URL patterns* in *Django*.

In Django, **URL patterns** direct web requests to the appropriate view for processing. They are defined in the `urls.py` file of each app and can be simple strings or regular expressions.

### Basic URL Patterns

- **Basic Syntax**: A URL pattern is a string matching the incoming request's path. For a match to occur, the URL pattern from `urls.py` must be identical to the request path.
- **Example**:

  ```
  from django.urls import path
  from . import views

  urlpatterns = [
      path('articles/2022/', views.article_2022),
  ]
  ```

  In this example, only requests to `/articles/2022/` will match.

### Regular Expressions for Advanced URL Patterns

For more advanced matching, Django allows the use of regular expressions.

- **Syntax**: Such patterns open with `^` and end with `$` to indicate the full path. This syntax gives flexibility in pattern matching.
- **Example**:

```
from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r'articles/(\d{4})/', views.article_year),
]
```

Here, requests to paths like `/articles/2022/` will match, and the year part will be captured and sent to the associated view.

### URL Pattern Helpers

Django provides several helper functions to streamline URL pattern definitions:

- **path()**: For simple matching based on the path.
- **re_path()**: For using regular expressions.
- **include()**: Enables URL grouping and delegation to other URL config files.

### Best Practices

#### Keep URLs Simple

For many projects, basic string matching in `path()` functions suffices. This approach enhances readability and maintainability.

#### Use Unambiguous URL Designs

Clear URL designs improve both developer and user experiences.

#### Leverage Named URLs

Assigning names to URLs via `path()` and `re_path()` aids in referencing URLs in templates, keeping code DRY (Don't Repeat Yourself).

## 11. What is a *database migration* in *Django* and why is it important?

**Database migrations** in Django are changes to the database schema, ensuring it stays in sync with the evolving data models. Each migration is represented as a **Python file**, making alterations via the `django.db.migrations` module.

### Key Components

- **Migration Files**: These are generated sequentially and outline the changes to be applied to the database.
- **Migrations Modules**: A collection of related migration files.
- **Migration Plan**: A record of applied and unapplied migrations.

### Why Use Migrations?

- **Version Control**: Migrations are text-based, making them ideal for version control systems like Git.
- **Data Integrity**: They help ensure that data remains consistent as the schema evolves.
- **Collaboration**: Simplifies team collaboration and deployment processes.
- **Adaptability**: Migrations can be tailored for different databases, such as PostgreSQL or MySQL.

### Basic Migration Commands

1. **Initial Migration**: Generate the initial database schema from models.

```
python manage.py makemigrations
```

2. **Apply Migrations**: Execute the pending migrations on the database.

```
python manage.py migrate
```

3. **Migrations Plan**: Check the status of migrations.

```
python manage.py showmigrations
```

Advanced developers may often require more intricate migration commands and functionalities. Fortuitously, Django offers a versatile array of options to accommodate those needs.

## 12. Explain the difference between a *ForeignKey* and a *ManyToManyField* in *Django models*.

In Django models, relationships can be established using either a **ForeignKey** or a **ManyToManyField**, each serving a unique purpose.

### ForeignKey

A **ForeignKey** sets up a **many-to-one** relationship. It's used when each record in the current model needs to be associated with exactly one record in another model.

**Example:**

- A `Book` has one `Publisher`.
- This is represented by a ForeignKey in the `Book` model pointing to the `Publisher` model.

**Code Example (ForeignKey):**

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
```

### ManyToManyField

A **ManyToManyField** is used when a record in one model can be associated with multiple records in another, and vice versa.

**Many-to-Many Relationship**

It facilitates a **many-to-many** relationship where multiple records in one model can be linked to multiple records in another.

**Example:**

- A `Student` can enroll in multiple `Course`s, and a `Course` can have multiple students enrolled.
- This many-to-many relationship is represented using a ManyToManyField with an intermediary table.

**Code Example (ManyToManyField):**

```
from django.db import models

class Course(models.Model):
    name = models.CharField(max_length=100)

class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField(Course)
```

**ManyToManyField with Additional Fields (through):**

Sometimes, the relationship between the two models needs to have additional fields. `ManyToManyField` allows you to specify a custom intermediary model using the `through` parameter.

Here is an example:

You have a Music Library and want to associate `Song`s with `Playlist`s, but you also want to track the order of the songs in each playlist.

You would use a model like `PlaylistSong` as the intermediary model.

```
class Playlist(models.Model):
    name = models.CharField(max_length=100)
```

```
class Song(models.Model):
    title = models.CharField(max_length=100)
    playlists = models.ManyToManyField(Playlist, through='PlaylistSong')

class PlaylistSong(models.Model):
    song = models.ForeignKey(Song, on_delete=models.CASCADE)
    playlist = models.ForeignKey(Playlist, on_delete=models.CASCADE)
    order = models.IntegerField()
    # Possibly other fields like 'added_by' and 'date_added'
```

Here, the `PlaylistSong` model contains the additional `order` field to track the song's order in the playlist.

## 13. How do you define a custom *model field* in *Django*?

In Django, you can define a custom model field to encapsulate complex data types, enforce specific behavior, or integrate with external data sources.

### Custom Model Field Definition

Here is the Python code:

```
from django.db import models
from django.core.exceptions import ValidationError

class MyCustomField(models.Field):
    description = "A custom field for special data handling."

    def __init__(self, *args, **kwargs):
        # Initialize your custom field attributes here
        super().__init__(*args, **kwargs)

    def db_type(self, connection):
        # Define the database column type
        return 'SOME_DB_TYPE'

    def from_db_value(self, value, expression, connection):
        # Convert the database value to the expected Python object
        if value is not None:
            # Perform any necessary data transformations or validations
            return transformed_value
        return value

    def to_python(self, value):
        # Convert the value to the appropriate Python object type
        if isinstance(value, str):
            return parse_string_value(value)
        return value

    def get_prep_value(self, value):
        # Convert the provided Python value for storage in the database
        return formatted_value

    def validate(self, value, model_instance):
        # Implement custom data validations
        if not is_valid:
            raise ValidationError("Invalid data.")

    def formfield(self, **kwargs):
        # Customize the form field for this model field
        defaults = {'form_class': CustomFormFieldClass}
        defaults.update(kwargs)
        return super().formfield(**defaults)
```

### Field Methods Overview

- **__init__**: Initializes custom field attributes.
- **db_type**: Specifies the database column type.
- **from_db_value**: Converts database value to Python object.
- **to_python**: Specifies the Python data type.
- **get_prep_value**: Converts value for storage in the database.
- **validate**: Provides custom data validations.

- **formfield**: Customizes the form field for the field type.

### Use Cases for Custom Fields

1. **Encapsulating Legacy Data**: Handle complex or legacy data formats transparently.
2. **External Data Integration**: Connect to external data sources or APIs for specific data requirements.
3. **Advanced Data Validation**: Implement custom or advanced data validation rules beyond what `validators` or `clean` methods offer.
4. **Specialized Data Types**: Manage special data types or structures not covered by standard Django fields.
5. **Embedding Business Logic**: Integrate data-specific business logic to be carried out at the model or form level.

## 14. What is a *QuerySet* in *Django* and how is it used?

In Django, a **QuerySet** represents a collection of **database queries** that can be **chained** and lazily executed. It provides an intuitive way to retrieve and manipulate data from the database using Python.

### QuerySet Basics

- **Initializationic**: QuerySets are generated automatically when you interact with a Django model using its Manager.
- **Chaining**: Multiple methods can be chained together before the QuerySet is evaluated.
- **Lazy Evaluation**: QuerySets are only executed when data is required, such as when iterating through the results or explicitly calling evaluation methods like `list()` or `count()`.
- **Immutability After Evaluation**: Once a QuerySet is evaluated, its results cannot be altered. For instance, you can't add more items to a list after it's been created.

### QuerySet Methods

#### Data Retrieval

- **all()**: Returns all objects in the QuerySet.
- **get()**: Retrieves a single object that matches the provided criteria.
- **filter()**: Returns a subset of objects that match specific criteria.
- **exclude()**: Returns objects that don't match the specified criteria.

#### Data Manipulation

- **create()**: Instantly creates a new object and saves it to the database.
- **update()**: Modifies objects in the database that match the given criteria.

#### Relationship Handling

- **select_related()**: Fetches related Many-To-One objects. Efficient for queries with a small number of related objects.
- **prefetch_related()**: Fetches the related objects for a Many-To-Many or reverse ForeignKey relationship.

#### QuerySet Execution

- **iterator()**: Fetches objects from the database one at a time, useful for large result sets.
- **get_or_create()**: Attempts to fetch an object from the database based on certain criteria and creates it if it doesn't exist.
- **earliest() and latest()**: Retrieve the first and last object, respectively, based on the specified field.

#### Aggregation and Metrics

- **aggregate()**: Performs an aggregate function (e.g., Count, Sum, Avg) over the items in the QuerySet.
- **count()**: Returns the number of items in the QuerySet.

#### QuerySet Evaluation

- **bool()**: Returns `True` if the QuerySet contains any results, `False` otherwise.
- **exists()**: Checks if there are any results that match the query.

#### QuerySet Execution

- **order_by()**: Orders the QuerySet results based on the specified field.

- **reverse()**: Reverses the original ordering of the QuerySet.

**Slicing and Pagination**

- **iterator()**: Fetches objects from the database one at a time, useful for large result sets.
- **first() and last()**: Retrieve the first or last object from the QuerySet.
- **count()**: Returns the number of items in the QuerySet.

**QuerySet Convenience Methods**

- **in_bulk()**: Returns a dictionary of objects with their primary keys as the keys.
- **values() and values_list()**: Return dictionaries or tuples, respectively, representing the objects' fields.
- **distinct()**: Removes duplicate results from the QuerySet.
- **union()**: Combines two or more QuerySets into a single, unique QuerySet.

## 15. Describe the concept of *model inheritance* in *Django*.

**Model inheritance** in Django allows you to create new models based on existing ones, leading to efficient code reuse and structured data management.

### Model Inheritance Types

1. **Abstract Base Classes Inheritance**: The parent model (defined as `abstract`) serves purely as a template and isn't used to create any database tables on its own.
2. **Multi-Table Inheritance**: This approach creates **separate database tables** for the parent model and each of its child models, maintaining a one-to-one relationship.

### Common Inheritance Patterns

1. **STI - Single Table Inheritance**
2. **MTI - Multi-Table Inheritance**
3. **Concrete Classes vs. Abstract Base Classes**

### Inheritance in Relational Databases

In relational databases, inheritance is handled via table relationships. Django supports both the STI and MTI strategies.

### Code Example: Abstract Base Class

Here is the Django code:

```
from django.db import models

class CommonInfo(models.Model):
    # This acts as an abstract base class.
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        # Set 'abstract' to True to make this an abstract base class.
        abstract = True


class Student(CommonInfo):
    # Inherits fields 'name' and 'age' from CommonInfo.
    student_id = models.CharField(primary_key=True, max_length=10)
```

The `CommonInfo` model here is an **abstract base class**. It's not directly instantiated in the database but provides fields like `name` and `age` for any model that inherits from it.

The `Student` model inherits from `CommonInfo` and therefore shares its fields in addition to its own, like `student_id`.

The generated SQL for `Student` will include fields `name`, `age`, and `student_id`.

### Code Example: Multi-Table Inheritance

Here is the Django code:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

This results in two tables: one for `Place` with fields `name` and `address`, and another for `Restaurant` with additional fields `serves_hot_dogs` and `serves_pizza`. The `Restaurant` table has a **one-to-one** relationship with the `Place` table.

**Explore all 70 answers here 👉 Devinterview.io - Django**



### Releases

No releases published

### Packages

No packages published