Ecom – Website

**🔹 Project Basics**

**Q1: What is this project about?**  
**A:** This is an e-commerce website built with Django. It allows users to browse products, search, view product details, and place orders. The admin interface is also customized for easier product and order management.

**🔹 Admin Customization**

**Q2: How have you customized the Django admin interface?**  
**A:** I customized the Django admin by:

* Changing the site header, title, and index title using admin.site.site\_header, site\_title, and index\_title.
* Creating a ProductAdmin class to control how products are displayed and edited.
* Implementing a custom action change\_category\_to\_defalut to set a product's category to "default".
* Adding search fields, list display, editable fields, and action functionalities to streamline admin workflows.

**Q3: What's the purpose of change\_category\_to\_defalut method?**  
**A:** It's a custom admin action that allows bulk updating of the product category to "default" directly from the admin panel. This helps in quickly categorizing multiple products.

**🔹 Pagination and Filtering**

**Q4: How did you implement pagination in your views?**  
**A:** I used Django’s built-in Paginator class in the index view. It paginates the list of products, displaying 4 items per page. The current page number is fetched using request.GET.get('page'), and the corresponding page object is passed to the template.

**Q5: How does search functionality work in your index view?**  
**A:** It uses a GET parameter called item\_name. If the parameter is provided, the queryset is filtered using title\_\_icontains=item\_name, which allows case-insensitive partial matching on product titles.

**🔹 Order Handling and Checkout**

**Q6: How do you handle order creation in the checkout view?**  
**A:** On a POST request to the checkout view, I collect customer details and the cart data from the form. The cart data is sent as a JSON string and then deserialized using json.loads. A new Order object is created and saved with this data.

**Q7: How do you ensure that the cart data is valid JSON?**  
**A:** I wrap json.loads(cart\_data) in a try...except block to catch and handle json.JSONDecodeError. If the JSON is invalid, I fallback to an empty dictionary.

**🔹 Other Django Concepts**

**Q8: What is get\_asgi\_application() used for?**  
**A:** get\_asgi\_application() is used to get the ASGI-compatible application object for serving the Django project using ASGI servers like Daphne or Uvicorn. It’s necessary for asynchronous features and deploying with ASGI.

**Q9: Why do you set the environment variable DJANGO\_SETTINGS\_MODULE in asgi.py?**  
**A:** It tells Django which settings module to use for configuring the application. It’s essential for Django to load the correct configurations when the application starts.

**🔹 Suggestions for Improvements**

**Q10: How would you improve this project further?**  
**A:**

* Add user authentication for order tracking and personalized experience.
* Implement cart persistence using sessions or database models.
* Add validation and success messages after checkout.
* Implement AJAX-based product filtering and cart updates for better UX.
* Add unit tests for views and models to improve code reliability.

**Q11: What is the difference between GET and POST in Django views?**  
**A:** GET is used for retrieving data (like displaying products), while POST is used to send data to the server (like submitting orders). In my checkout view, I handle POST to save order data.

**Q12: How does Django handle form submission in the checkout view?**  
**A:** I use request.POST.get() to extract form data and then manually create an Order object. In a full implementation, I could also use Django Forms for built-in validation.

**Q13: What would happen if the product ID doesn’t exist in the detail view?**  
**A:** It would raise a DoesNotExist error. To handle it gracefully, I can use get\_object\_or\_404(Products, id=id) instead of Products.objects.get(id=id).

**Q14: What are some Django security practices you’ve followed or would add?**  
**A:**

* CSRF protection via Django's built-in middleware
* Using json.loads with try-except to avoid crashes
* Input validation in forms
* Session-based cart or authentication for secure checkout

**Q15: What is the role of templates in your project?**  
**A:** Templates render dynamic HTML pages by injecting data from views (like product lists, details). I use 'shop/index.html', 'shop/detail.html', and 'shop/checkout.html' to create the user interface.

Web Scraper

## ✅ Project Overview

### ****Q1: What does your web scraper project do?****

**A:** It allows users to input a website URL, fetches the webpage, extracts all the anchor (<a>) tags using BeautifulSoup, and stores the link text and address into a database using Django's ORM. Users can also delete individual or all links via buttons on the interface.

## ✅ Questions on Packages and Code Logic

### ****Q2: What is the use of**** requests ****in your project?****

**A:** The requests library is used to make HTTP GET requests to the user-provided URL. It retrieves the HTML content of the page so it can be parsed for links.

### ****Q3: Why do you use**** verify=False ****in**** requests.get()****?****

**A:** This disables SSL certificate verification, which helps avoid errors when scraping sites with expired or self-signed certificates. However, it’s **not recommended in production** due to security risks. In a real deployment, I would use verify=True.

### ****Q4: What is BeautifulSoup and how do you use it?****

**A:** BeautifulSoup is a Python library used to **parse HTML or XML documents**. In my project, I use it to parse the HTML content returned from the requests.get() call:

python

CopyEdit

soup = BeautifulSoup(page.text, 'html.parser')

Then I extract all anchor tags with:

python

CopyEdit

soup.find\_all('a', href=True)

It helps in navigating and searching through HTML elements efficiently.

### ****Q5: What is**** urljoin ****used for in your project?****

**A:** Many websites use **relative URLs** (like /about), so I use urljoin(site, href) to convert them to **absolute URLs** (like https://example.com/about). This ensures all links are valid and complete.

### ****Q6: What is the purpose of**** Link.objects.create(...)****?****

**A:** It saves each scraped link into the database using Django’s ORM. Each link has two fields: name (the visible text) and address (the actual URL).

### ****Q7: What does**** Link.objects.last() ****do?****

**A:** It returns the **most recently added** Link object from the database. I use it when the user clicks "Delete Last" to remove the latest entry.

### ****Q8: Why do you use**** HttpResponseRedirect('/') ****instead of**** render() ****after POST?****

**A:** It's a common best practice called **Post/Redirect/Get pattern (PRG)**. It prevents form resubmission if the user refreshes the page after submitting a form. HttpResponseRedirect('/') redirects the user to the homepage after scraping or deleting.

### ****Q9: Why do you use**** print() ****statements in your scraper?****

**A:** For debugging during development. They help verify what links are being scraped. In production, logging should be used instead of print statements.

### ****Q10: How do you handle errors like bad URLs or failed requests?****

**A:** I wrap the requests.get() call in a try-except block to catch requests.exceptions.RequestException. If an error occurs (e.g., invalid URL or timeout), it prints the error without crashing the app.

## ✅ Model & Database

### ****Q11: What does the**** \_\_str\_\_ ****method do in your**** Link ****model?****

**A:** It defines how each Link object is displayed in Django admin or shell. I return the name of the link, or "Unnamed Link" if the name is empty.

### ****Q12: Why did you use**** blank=True, null=True ****in your model fields?****

**A:** This allows fields to be optional. null=True allows NULL values in the database, and blank=True allows the Django forms to accept empty inputs.

## ✅ Additional / Advanced Questions

### ****Q13: How would you improve this project?****

**A:**

* Add validation to check if the URL is valid before scraping.
* Store only unique links to avoid duplication.
* Display error or success messages in the UI.
* Add user authentication to make link data user-specific.
* Use Django messages framework for user feedback.

### ****Q14: How can you avoid scraping duplicate links?****

**A:** Before saving, I can check if the link already exists in the database using:

python

CopyEdit

if not Link.objects.filter(address=link\_address).exists():

Link.objects.create(...)

### ****Q15: What are some ethical considerations when web scraping?****

**A:** It’s important to:

* Respect the website’s robots.txt.
* Avoid scraping content behind login or paywalls.
* Not overload the server with too many requests (use rate limiting).
* Give proper credit or usage according to the site's terms of service.

## 🧠 Summary of Tools Used

| **Tool / Library** | **Purpose** |
| --- | --- |
| **requests** | Fetches HTML from the web |
| **BeautifulSoup** | Parses HTML to extract elements |
| **urllib.parse.urljoin** | Converts relative URLs to absolute URLs |
| **urllib3.disable\_warnings** | Temporarily suppresses SSL warnings (for dev only) |
| **Django Models (Link)** | Stores and manages scraped links |
| **HttpResponseRedirect** | Implements PRG pattern after POST |
| **Django ORM** | Saves and retrieves link objects from the DB |

Expense Tracker

## ✅ Project Overview

### ****Q1: What is your Expense Tracker project about?****

**A:** It’s a Django-based web application where users can add, view, edit, and delete expenses. It also provides summaries (weekly, monthly, yearly) and visualizes expenses by date and category using graphs generated with matplotlib.

## ✅ Core Functionality

### ****Q2: How do you calculate the total, weekly, monthly, and yearly expenses?****

**A:** I use Django’s ORM and datetime to filter expenses:

python

CopyEdit

last\_year = datetime.date.today() - datetime.timedelta(days=365)

yearly\_expense = Expense.objects.filter(date\_\_gt=last\_year).aggregate(Sum('amount'))

Similar logic is used for monthly and weekly expenses using timedelta(30) and timedelta(7).

### ****Q3: How are the expenses visualized?****

**A:** Using matplotlib, I generate two graphs:

* **Daily Expense Graph**: Line chart showing how much was spent each day.
* **Category Expense Graph**: Line chart showing total spend per category.

The charts are rendered in memory using io.BytesIO() and encoded with base64 to embed them in HTML.

### ****Q4: How is data saved in the application?****

**A:** Data is saved through a Django ModelForm called ExpenseForm. When a user submits the form, it’s validated and saved using Django's ORM.

### ****Q5: How do you handle expense editing and deleting?****

**A:**

* **Editing** uses a form prefilled with the instance via:

python

CopyEdit

ExpenseForm(instance=expense)

* **Deleting** is done using Expense.objects.get(id=id).delete() after a POST request.

## ✅ Code & Structure

### ****Q6: What does the**** generate\_plot() ****function do?****

**A:** It generates a matplotlib line chart from input data and labels, styles it with titles, markers, and axis labels, and returns a base64 string of the image that can be directly embedded into a Django template using <img> tags.

### ****Q7: Why do you use**** aggregate(Sum('amount')) ****instead of looping?****

**A:** aggregate() performs the sum at the database level, which is more efficient than looping through results in Python. It reduces memory usage and improves performance.

### ****Q8: Why do you use**** auto\_now=True ****in the**** date ****field?****

**A:** It automatically sets the current date every time the record is saved. Though it’s good for this use case, auto\_now\_add=True might be better if I want to keep the original creation date unchanged.

## ✅ Design & UX

### ****Q9: How does your app handle invalid form submissions?****

**A:** Django forms automatically handle validation. If the form is invalid, it doesn’t save the data and can display errors in the template.

### ****Q10: What are some improvements you’d like to make?****

**A:**

* Add **user login and session-based expenses**.
* Convert graphs to **bar or pie charts** for better category comparison.
* Add **filters** for date ranges or categories.
* Export data to **CSV or Excel**.

## ✅ Extra/Advanced Questions

### ****Q11: What are the advantages of using ModelForm?****

**A:** ModelForm automatically links a Django model to the form, reducing repetitive code and handling both validation and saving.

### ****Q12: How would you make this a multi-user system?****

**A:** Add a ForeignKey to User in the Expense model:

python

CopyEdit

from django.contrib.auth.models import User

user = models.ForeignKey(User, on\_delete=models.CASCADE)

Then filter/query expenses by the currently logged-in user.

### ****Q13: Why do you use**** get\_object\_or\_404()****?****

**A:** It’s safer than Expense.objects.get() because it returns a 404 page if the object doesn’t exist, avoiding app crashes.

## 🧠 Summary of Tools & Concepts Used

| **Tool / Concept** | **Purpose** |
| --- | --- |
| Django ORM | For saving and querying expense data |
| ModelForm | Automatically builds forms from models |
| Matplotlib | For generating expense graphs |
| base64 + BytesIO | To embed graphs in HTML without saving images |
| aggregate(Sum(...)) | Efficiently calculate totals in the database |
| datetime | Time-based filtering (7 days, 30 days, etc.) |

**Q1: How would you normalize the Expense model (e.g., split category into its own table)?**

**Answer:**

To normalize the Expense model, I would extract the category field into a separate model and use a ForeignKey relationship. This reduces data duplication and ensures category consistency.

**Updated models:**

python

CopyEdit

class Category(models.Model):

name = models.CharField(max\_length=100, unique=True)

def \_\_str\_\_(self):

return self.name

class Expense(models.Model):

name = models.CharField(max\_length=100)

amount = models.IntegerField()

category = models.ForeignKey(Category, on\_delete=models.CASCADE)

date = models.DateField(auto\_now=True)

**Benefits:**

* Data integrity (no typos or inconsistencies in categories)
* Easier filtering, grouping, and analytics
* Dynamic category management

**💡 Q2: How would you implement pagination or search for large expense lists?**

**Answer:**

For large datasets, I would use Django's built-in Paginator class to break the expense list into manageable pages. I’d also use a simple keyword filter for expense names or categories.

**Example code:**

python

CopyEdit

from django.core.paginator import Paginator

def index(request):

query = request.GET.get('q')

expenses = Expense.objects.all()

if query:

expenses = expenses.filter(name\_\_icontains=query)

paginator = Paginator(expenses, 10) # 10 per page

page\_number = request.GET.get('page')

page\_obj = paginator.get\_page(page\_number)

return render(request, 'expensetracter/index.html', {'page\_obj': page\_obj})

**Bonus tip**: For better UX, add a search bar and next/prev buttons in your HTML.

**💡 Q3: How would you make each user’s expenses private (user authentication)?**

**Answer:**

I’d use Django’s built-in User model and add a ForeignKey to link each expense to the authenticated user. This way, each user can only see and manage their own data.

**Model update:**

python

CopyEdit

from django.contrib.auth.models import User

class Expense(models.Model):

user = models.ForeignKey(User, on\_delete=models.CASCADE)

name = models.CharField(max\_length=100)

amount = models.IntegerField()

category = models.ForeignKey(Category, on\_delete=models.CASCADE)

date = models.DateField(auto\_now=True)

**Views:**

python

CopyEdit

def index(request):

if request.user.is\_authenticated:

expenses = Expense.objects.filter(user=request.user)

...

**Why this is good:**

* Protects user data
* Prepares app for multi-user environments

**💡 Q4: If asked about Class-Based Views (CBVs), what would you say?**

**Answer:**

I started with function-based views for simplicity, but I would use class-based views for larger projects. CBVs improve code reusability and follow DRY principles. For example, I’d use ListView for listing expenses and CreateView for adding them.

**Example (CBV for listing expenses):**

python

CopyEdit

from django.views.generic import ListView

from .models import Expense

class ExpenseListView(ListView):

model = Expense

template\_name = 'expensetracter/index.html'

context\_object\_name = 'expenses'

def get\_queryset(self):

return Expense.objects.filter(user=self.request.user)

**Benefits:**

* Less repetitive code
* Cleaner structure with built-in support for pagination, filtering, etc.

### ✅ ****Tailwind CSS Interview Questions & Answers****

#### **1. What is Tailwind CSS and how is it different from Bootstrap?**

Tailwind is a utility-first CSS framework where you build custom designs using low-level utility classes. Unlike Bootstrap, which provides pre-styled components, Tailwind offers full control over your design with minimal custom CSS.

#### **2. What do you mean by utility-first CSS?**

Utility-first means using classes like p-4, text-center, or bg-blue-500 directly in HTML to style elements, instead of writing separate CSS rules.

#### **3. What are the benefits of using Tailwind CSS in your projects?**

Speed, flexibility, responsive design out of the box, and no need to switch between CSS and HTML. It also improves design consistency using a standardized design system (spacing, colors, etc.).

#### **4. How do you manage long class names in Tailwind HTML?**

I use reusable components with Django templates ({% include %} or {% block %}), and sometimes extract styles using Tailwind’s @apply directive in a custom CSS file.

#### **5. How did Tailwind help in building responsive layouts?**

Tailwind uses a mobile-first approach with breakpoints like sm:, md:, lg:. For example, I used grid-cols-1 md:grid-cols-2 to switch from single-column to two-column layout on tablets or desktops.

#### **6. How did you use Tailwind in your Django project setup?**

I used Tailwind via PostCSS and tailwindcss-cli for rapid builds. I integrated it into Django templates with static file linking and built Tailwind components directly in the HTML.

#### **7. How did you customize your Tailwind configuration?**

I edited tailwind.config.js to add custom colors, fonts, spacing, and shadows for consistent branding across my app.

#### **8. Can you give examples of utilities you used frequently?**

Yes, I used:

* Layout: flex, grid, gap-4, items-center
* Spacing: p-4, m-2, space-y-4
* Typography: text-lg, font-semibold
* Colors: bg-gray-100, text-blue-600, hover:bg-blue-700
* Effects: rounded-lg, shadow-md, transition

#### **9. How do you add hover effects or transitions in Tailwind?**

Tailwind provides built-in hover utilities like hover:bg-blue-500. You can add transition, duration-300, and ease-in-out for smooth hover effects.

#### **10. How does Tailwind handle dark mode?**

Tailwind supports dark mode using the dark: prefix. You can enable it in the config file and use utilities like dark:bg-gray-800, dark:text-white.

#### **11. What is the @apply directive and when do you use it?**

@apply lets you group utility classes into custom CSS rules inside a stylesheet. I use it when class names are repeated or getting too long.

#### **12. How do you handle form styling in Tailwind?**

Tailwind gives basic styling to form elements. I used input, label, select elements with classes like border, rounded, p-2, focus:outline-none to build clean forms.

#### **13. How do you optimize Tailwind CSS for production?**

I use Tailwind’s purge feature (now called content) in tailwind.config.js to remove unused CSS and reduce file size. For example:

js

CopyEdit

content: ['./templates/\*\*/\*.html', './static/\*\*/\*.js']

#### **14. What was the biggest challenge you faced using Tailwind and how did you solve it?**

Initially, class clutter in templates was a challenge. I solved it by reusing components with Django templates and using @apply for repetitive utility patterns.

#### **15. How did Tailwind improve your overall development workflow?**

It reduced the need to write custom CSS, allowed rapid prototyping, and helped maintain design consistency. Also, working within a single file improved my productivity during frontend tasks.

**💡 Q1: Why did you choose Tailwind CSS over other frameworks like Bootstrap?**

**Answer:**

I chose Tailwind CSS because it gives me more control over the design without forcing predefined components like Bootstrap. It's utility-first, so I can build custom UIs quickly while keeping the HTML structure semantic and lightweight. Also, it integrates well with Django templates and promotes consistency in spacing, color, and layout.

**💡 Q2: How did Tailwind improve your UI development process?**

**Answer:**

Tailwind improved my workflow by reducing the need for custom CSS. I could prototype and iterate designs much faster. For example, building responsive cards for listing expenses with hover effects and grid layouts took just a few lines of class-based markup, without switching between HTML and CSS files.

**💡 Q3: How do you handle responsiveness with Tailwind?**

**Answer:**

Tailwind makes responsiveness easy with its mobile-first breakpoints like sm:, md:, lg: etc. I used them in my project to ensure expense lists, charts, and forms look good on both desktop and mobile. For instance, I used grid grid-cols-1 md:grid-cols-2 to switch between single-column and two-column layouts based on screen size.

**💡 Q4: Did you customize the Tailwind config file?**

**Answer:**

Yes, I customized the tailwind.config.js file to add custom colors and spacing for consistent branding across my project. I also extended the font family and added shadows for card components to improve the visual feel.

**💡 Q5: How do you keep your HTML clean when using Tailwind?**

**Answer:**

I use Django template inheritance to avoid duplication and keep components reusable. For example, I created reusable components like buttons and forms using {% include %} and Tailwind utility classes. When things get too messy, I extract the class sets into small template snippets or use @apply inside custom CSS files.

**💡 Q6: What are some Tailwind utilities you used most in your project?**

**Answer:**

Some of the most-used utilities were:

* flex, grid, gap-x, gap-y for layout
* bg-gray-100, text-blue-600, hover: for styling
* rounded-lg, shadow-md for cards
* w-full, max-w-md for form responsiveness
* transition, duration-300 for smooth effects

Django – 3Year

#### 1. **What is Django?**

**Answer:**  
Django is a high-level Python web framework that enables developers to create robust, scalable web applications quickly. It follows the Model-View-Template (MVT) architecture and emphasizes reusability, rapid development, and the "don't repeat yourself" (DRY) principle.

#### 2. **What is the difference between Django and Flask?**

**Answer:**  
Django is a full-fledged web framework that comes with built-in features like authentication, ORM (Object-Relational Mapping), admin panel, and more. Flask, on the other hand, is a micro-framework that provides the basics, giving developers more flexibility to choose other libraries and tools for building web applications.

#### 3. **What are Django models?**

**Answer:**  
Django models are Python classes that define the structure of your database. They map to a database table, with each attribute of the model representing a column in the table. Models handle database operations, allowing developers to interact with the database using Python objects.

#### 4. **Explain the purpose of Django’s settings.py file.**

**Answer:**  
The settings.py file contains configuration settings for a Django project. It defines various settings such as database configurations, middleware, static files settings, app configurations, security settings, and more.

### Intermediate Questions

#### 5. **What are Django migrations and how do they work?**

**Answer:**  
Migrations are a way to apply changes made to the models (such as adding or removing fields) to the database schema. When you modify a model, you create a migration file that Django uses to apply these changes to the database. Migrations ensure that the database schema stays in sync with your models.

You can create migrations using python manage.py makemigrations, and apply them using python manage.py migrate.

#### 6. **What is the purpose of Django admin?**

**Answer:**  
The Django admin is an automatically-generated web interface that allows developers and site administrators to manage the data in the application. It’s built using the models defined in Django, and it provides an intuitive, customizable way to interact with the application's database.

#### 7. **What are Django views, and how are they used?**

**Answer:**  
Views in Django are Python functions or classes that handle HTTP requests and return HTTP responses. A view receives input from a user (usually from a URL), processes it (e.g., by querying the database), and returns a response, often rendering a template or returning data (e.g., JSON).

#### 8. **Explain what middleware is in Django.**

**Answer:**  
Middleware in Django is a layer of processing that occurs during the request-response cycle. It is a framework for processing requests globally before they reach views or after the response has been returned from views. Middleware can be used for things like session management, user authentication, or modifying request/response objects.

### Advanced Questions

#### 9. **What is the difference between a Django model’s** save() **and** update() **method?**

**Answer:**

* save(): This method is used to create a new instance or update an existing instance in the database. It performs both insert and update operations, depending on whether the instance exists in the database.
* update(): This method is typically used to update existing records in the database, and it does not perform any insert operation.

#### 10. **Explain Django’s ORM (Object-Relational Mapping).**

**Answer:**  
Django’s ORM allows developers to interact with the database using Python objects instead of writing raw SQL queries. The ORM translates Python code into SQL commands behind the scenes. It simplifies database operations like querying, inserting, updating, and deleting records, allowing developers to work with database models as objects.

#### 11. **How does Django handle security?**

**Answer:**  
Django provides several built-in security features:

* **CSRF protection**: Protects against Cross-Site Request Forgery attacks.
* **XSS protection**: Automatically escapes data to prevent cross-site scripting attacks.
* **SQL Injection protection**: Django’s ORM uses parameterized queries to protect against SQL injection.
* **Password hashing**: Django uses strong hashing algorithms to store user passwords securely.
* **Clickjacking protection**: Prevents clickjacking by using an HTTP header.

#### 12. **What is the purpose of** select\_related() **and** prefetch\_related() **in Django ORM?**

**Answer:**

* select\_related(): It performs a SQL join and includes the related objects in the query result. It is used for ForeignKey and OneToOne relationships and is generally more efficient in retrieving related objects.
* prefetch\_related(): It performs separate queries for each relationship and does the "joining" in Python. It is used for ManyToMany and reverse ForeignKey relationships and is useful when dealing with many-to-many relationships.

#### 13. **Explain Django’s request/response cycle.**

**Answer:**  
Django follows a request-response cycle:

1. A user makes an HTTP request.
2. The request is passed through middleware for processing (e.g., authentication, logging).
3. Django matches the request to a view based on the URL pattern.
4. The view handles the request, interacting with the database and preparing a response (often rendering a template).
5. The response is passed through middleware (e.g., to set cookies, headers).
6. The response is returned to the user.

#### 14. **How would you optimize a Django application for performance?**

**Answer:**

* **Database optimization**: Use select\_related() and prefetch\_related() to minimize the number of queries.
* **Caching**: Cache expensive views or data to avoid repeated processing (using Django’s caching framework).
* **Database indexing**: Add appropriate indexes to frequently queried columns.
* **Query optimization**: Avoid N+1 query problems and use efficient queries.
* **Static files**: Use a content delivery network (CDN) to serve static files and minimize server load.
* **Asynchronous tasks**: Offload long-running tasks to background workers using Celery.

#### 15. **How does Django handle file uploads?**

**Answer:**  
Django provides built-in support for handling file uploads. When a file is uploaded, Django stores it in a directory specified in the MEDIA\_ROOT setting. The FileField or ImageField is used in a model to define fields for storing uploaded files. When a user uploads a file, it can be accessed via the request.FILES object.

### Expert Questions

#### 16. **What are Django signals, and how are they used?**

**Answer:**  
Django signals allow decoupled applications to get notified when certain actions occur in the application (e.g., when a model is saved or deleted). They are useful for triggering events automatically. For example, the post\_save signal is sent after a model instance is saved.

#### 17. **Explain Django’s transaction management.**

**Answer:**  
Django provides a transaction management system that allows for atomic database operations. If something goes wrong in a set of database operations, Django ensures that all operations are rolled back to maintain consistency. The @transaction.atomic decorator is used to wrap code in a transaction, ensuring that the changes are committed only if all operations are successful.

#### 18. **How does Django’s** auth **system work?**

**Answer:**  
Django’s auth system provides built-in views, models, and forms for managing user authentication, including login, logout, password management, and permissions. It includes a User model that can be extended to handle custom user attributes. Authentication is typically done using a session-based system.

#### 19. **What is Django Rest Framework (DRF)?**

**Answer:**  
Django Rest Framework is a powerful toolkit for building Web APIs in Django. It simplifies the process of creating RESTful APIs, including serializing data, authentication, permission management, and view handling. DRF provides tools like APIView, ViewSet, and Serializer to create and handle API endpoints.

#### 20. **What are some ways to scale Django applications?**

**Answer:**

* **Database scaling**: Use database replication and sharding to distribute load.
* **Caching**: Implement caching mechanisms (e.g., Memcached or Redis) to reduce database load.
* **Load balancing**: Use load balancers to distribute traffic across multiple application servers.
* **Asynchronous processing**: Use task queues like Celery to offload long-running tasks.
* **Vertical scaling**: Increase the server’s resources (CPU, RAM) to handle more requests.

 **Deployment & Environment Setup:**

* **How do you deploy a Django application?**  
  **Answer:** Deployment of a Django app often involves setting up a web server (e.g., Nginx or Apache) with a WSGI server (e.g., Gunicorn). Static files are served by Nginx, and the application is connected to a database (e.g., PostgreSQL). You also configure environment variables, security settings, and may use Docker for containerization.
* **What is the purpose of ALLOWED\_HOSTS in Django?**  
  **Answer:** ALLOWED\_HOSTS is a security setting in Django that defines a list of trusted domains or IP addresses that can access the application. This prevents HTTP Host header attacks.

 **Testing:**

* **How do you write unit tests in Django?**  
  **Answer:** In Django, you can use TestCase from django.test to write unit tests for models, views, and other components. Django provides a test client to simulate HTTP requests and check the responses, and you can use fixtures to set up test data.
* **What is the difference between assertEqual() and assertContains() in Django tests?**  
  **Answer:** assertEqual() checks that two values are equal, while assertContains() checks that a response contains a particular substring (useful for checking if certain text appears in the rendered HTML).

 **Caching:**

* **What caching strategies does Django support?**  
  **Answer:** Django supports several caching strategies, including per-view caching (cache\_page), template fragment caching, and low-level caching (using cache backends like Memcached or Redis). There is also support for database caching, file-based caching, and others.

 **Advanced Django ORM:**

* **What is the Q object in Django, and how is it used?**  
  **Answer:** Q objects are used for complex queries, allowing you to use logical operators like AND, OR, and NOT. They enable more flexible filtering than just using keyword arguments in filter().

 **Advanced Security:**

* **What measures would you take to secure a Django app against SQL injection?**  
  **Answer:** Django’s ORM inherently protects against SQL injection by using parameterized queries. However, you can further secure your app by validating and sanitizing user input, avoiding raw SQL queries when possible, and using Django’s ORM for database operations.

 **Django REST Framework (DRF) Advanced:**

* **How do you handle authentication in Django Rest Framework?**  
  **Answer:** DRF provides several authentication classes, including TokenAuthentication, SessionAuthentication, and custom authentication classes. You can also use third-party packages like OAuth2 for more advanced authentication methods.
* **What are serializers in DRF, and how do they differ from Django models?**  
  **Answer:** Serializers in DRF are similar to Django models but are used to convert complex data types (such as model instances or querysets) into native Python datatypes (like dictionaries) that can be easily rendered into JSON, XML, or other content types. They are crucial for handling data in APIs.

 **Django Settings for Production:**

* **What are some key Django settings you would configure for production?**  
  **Answer:** Key settings for production include setting DEBUG = False, configuring ALLOWED\_HOSTS, setting up proper logging, using a production database (e.g., PostgreSQL), configuring STATIC\_URL and MEDIA\_URL, using SSL for security, and configuring caching and security middleware.

 **Asynchronous Support in Django:**

* **How does Django handle asynchronous views and tasks?**  
  **Answer:** Django 3.1+ introduced support for asynchronous views, middleware, and database queries. You can define asynchronous views using the async def syntax, and Django can use asynchronous tasks with packages like Celery.

**🔁 1. When should you use select\_related() vs prefetch\_related()?**

**Answer:**

* Use select\_related() for **ForeignKey** or **OneToOne** relationships. It performs a **SQL join** and fetches related objects in a single query.
* Use prefetch\_related() for **ManyToMany** or **reverse ForeignKey**. It performs **separate queries** and does the join in Python.

**Pattern:**

python

CopyEdit

books = Book.objects.select\_related('author') # For ForeignKey

authors = Author.objects.prefetch\_related('books') # For reverse FK

**🚫 2. What is the N+1 query problem in Django, and how do you avoid it?**

**Answer:**  
N+1 queries occur when you access related data in a loop, causing one query for the initial data and one extra query for each related object.

**Anti-pattern:**

python

CopyEdit

for book in Book.objects.all():

print(book.author.name) # Triggers one query per book

**Fix:** Use select\_related():

python

CopyEdit

for book in Book.objects.select\_related('author'):

print(book.author.name)

**🏗️ 3. What is the fat model-thin view pattern in Django?**

**Answer:**  
A best practice that suggests putting **business logic in models** or managers instead of views, to make views lightweight and models reusable.

**Example:**

python

CopyEdit

class Order(models.Model):

def total\_price(self):

return self.unit\_price \* self.quantity

**🔄 4. What is the use of custom managers in Django?**

**Answer:**  
Custom managers encapsulate common query logic and make models cleaner and DRY.

**Example:**

python

CopyEdit

class PublishedManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset().filter(status='published')

class Post(models.Model):

objects = models.Manager() # default

published = PublishedManager()

**🧹 5. What are common anti-patterns in Django development?**

**Answer:**

* Writing business logic in views (not models or forms)
* Using raw SQL unnecessarily
* Using filter().first() instead of get()
* Not using indexes on frequently queried fields
* Returning too much data in APIs

**🛡️ 6. What are some security best practices in Django?**

**Answer:**

* Set DEBUG = False in production
* Use ALLOWED\_HOSTS
* Sanitize user inputs
* Enable SECURE\_BROWSER\_XSS\_FILTER and X\_FRAME\_OPTIONS
* Use Django's CSRF protection and form validation
* Store secrets in environment variables

**🧪 7. How do you test Django views and models?**

**Answer:** Use TestCase from django.test. For views, use the test client to make requests. For models, use assertions to check business logic.

python

CopyEdit

def test\_order\_total\_price(self):

order = Order(unit\_price=100, quantity=2)

self.assertEqual(order.total\_price(), 200)

**🧠 8. What is the difference between get(), filter(), and get\_or\_create()?**

**Answer:**

* get(): Returns one object, raises DoesNotExist if not found.
* filter(): Returns a queryset; empty if no match.
* get\_or\_create(): Tries to get an object, creates it if it doesn’t exist.

**⚙️ 9. What is the use of @property in Django models?**

**Answer:** Used to define **read-only computed fields** in models that aren’t stored in the database.

python

CopyEdit

@property

def full\_name(self):

return f"{self.first\_name} {self.last\_name}"

**🧰 10. When would you use signals in Django?**

**Answer:** Use Django signals (like post\_save, pre\_delete) for **decoupled** event handling—e.g., send a welcome email after user registration.

Use sparingly. Prefer explicit function calls when logic is directly tied to app behavior.

**🚀 11. How would you scale a Django app for high traffic?**

**Answer:**

* Use load balancers and multiple app servers
* Implement caching (Redis/Memcached)
* Use Celery for background jobs
* Optimize DB with indexing and read replicas
* Serve static/media via CDN

**🪄 12. How do you implement custom permissions in Django?**

**Answer:** Define methods like has\_perm or use the @permission\_required decorator or PermissionRequiredMixin. You can also add custom permissions in the model's Meta.

python

CopyEdit

class Meta:

permissions = [

("can\_view\_reports", "Can view reports"),

]

**🗃️ 13. How do you use annotations and aggregations in Django ORM?**

**Answer:** Use .annotate() for adding computed fields, and .aggregate() for summary values.

python

CopyEdit

from django.db.models import Count

authors = Author.objects.annotate(num\_books=Count('books'))

**📦 14. What are reusable Django apps, and how do you create them?**

**Answer:** Reusable apps are Django apps that can be reused across projects (like django-crispy-forms). You can make your app reusable by avoiding hardcoded references, defining clear models, and writing pluggable URLs, templates, and settings.

**⛔ 15. What happens if you don’t set null=True for nullable fields?**

**Answer:** Django will use default behavior: null=False, and it will **raise an error** when trying to save None into non-nullable fields.

Use null=True for DB-level NULLs, blank=True for form validation.

**📈 16. What tools can you use to profile Django performance?**

**Answer:**

* Django Debug Toolbar
* Silk
* cProfile
* New Relic (for production monitoring)
* QuerySet .explain() for query planning

**🌐 17. What is the difference between STATICFILES\_DIRS and STATIC\_ROOT?**

**Answer:**

* STATICFILES\_DIRS: List of locations where Django looks for static files in dev mode.
* STATIC\_ROOT: The directory where collectstatic places all static files for production.

**🎯 18. How does Form.clean() differ from Field.clean()?**

**Answer:**

* Field.clean(): Validates individual field value.
* Form.clean(): Validates the entire form, used for interdependent field validation.

python

CopyEdit

def clean(self):

cleaned\_data = super().clean()

if cleaned\_data['password'] != cleaned\_data['confirm\_password']:

raise forms.ValidationError("Passwords do not match")

**🧵 19. How would you handle long-running tasks in Django?**

**Answer:** Use **Celery** with a broker like Redis or RabbitMQ. Offload tasks like sending emails, PDF generation, or data processing to workers.

**🕸️ 20. What is the role of ASGI in modern Django apps?**

**Answer:** ASGI (Asynchronous Server Gateway Interface) is the successor to WSGI and enables **async features** in Django, such as WebSockets and background tasks. Django 3.0+ supports ASGI, allowing developers to build real-time features like chat or live notifications.

### 1. ****What is Celery, and why would you use it in a Django project?****

**Answer:**  
Celery is an **asynchronous task queue** based on distributed message passing. It's used to handle background tasks outside the request/response cycle—like sending emails, data processing, or generating reports. It helps **improve performance** by offloading heavy or delayed jobs.

### 2. ****How do you integrate Celery into a Django project?****

**Answer:**

1. Install with pip install celery
2. Create a celery.py in your Django project folder:

python

CopyEdit

import os

from celery import Celery

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'myproject.settings')

app = Celery('myproject')

app.config\_from\_object('django.conf:settings', namespace='CELERY')

app.autodiscover\_tasks()

1. In \_\_init\_\_.py:

python

CopyEdit

from .celery import app as celery\_app

\_\_all\_\_ = ['celery\_app']

1. Define tasks in tasks.py:

python

CopyEdit

from celery import shared\_task

@shared\_task

def send\_email\_task():

# logic here

pass

### 3. ****What does Redis do in a Celery setup?****

**Answer:**  
Redis acts as a **message broker**. It temporarily stores task data and manages the communication between Django and Celery workers. You can also use it for **result storage** or caching.

## 🔌 **Django Channels (WebSockets & Real-Time)**

### 4. ****What is Django Channels, and when should you use it?****

**Answer:**  
Django Channels adds **WebSocket** and **asynchronous support** to Django. Use it for **real-time features** like chat apps, notifications, or live dashboards.

### 5. ****How do Channels differ from traditional Django views?****

**Answer:**  
Traditional Django views are **synchronous** (WSGI). Channels enable **asynchronous handling** via ASGI. Instead of HttpRequest, you get a **WebSocket consumer** that manages connect, receive, and disconnect events.

### 6. ****How do you create a WebSocket consumer using Channels?****

**Answer:**

python

CopyEdit

from channels.generic.websocket import AsyncWebsocketConsumer

import json

class ChatConsumer(AsyncWebsocketConsumer):

async def connect(self):

await self.accept()

async def receive(self, text\_data):

await self.send(text\_data=json.dumps({'message': 'Received!'}))

async def disconnect(self, close\_code):

pass

## 🐳 **Docker + Django**

### 7. ****Why would you Dockerize a Django app?****

**Answer:**  
Docker ensures consistent **development and production environments**. It simplifies dependencies, deployments, and scaling.

### 8. ****What does a basic Dockerfile for Django look like?****

**Answer:**

dockerfile

CopyEdit

FROM python:3.10

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000"]

### 9. ****What is the purpose of**** docker-compose ****in Django projects?****

**Answer:**  
docker-compose allows you to define **multi-container** setups—like Django + Postgres + Redis + Celery—in one YAML file, simplifying orchestration.

## ⚖️ **API Rate Limiting (Throttling)**

### 10. ****How do you add rate limiting to a Django Rest Framework API?****

**Answer:** Set throttling in settings.py:

python

CopyEdit

REST\_FRAMEWORK = {

'DEFAULT\_THROTTLE\_CLASSES': [

'rest\_framework.throttling.UserRateThrottle',

],

'DEFAULT\_THROTTLE\_RATES': {

'user': '100/hour',

}

}

You can also create a **custom throttle class** by subclassing BaseThrottle.

### 11. ****Why is API rate limiting important?****

**Answer:**  
Rate limiting prevents **abuse, spam, and server overload**. It ensures fair usage and protects your API from brute-force or DoS attacks.

## ⚙️ **GraphQL with Django (Graphene)**

### 12. ****What is GraphQL, and how does it differ from REST?****

**Answer:**  
GraphQL is a query language for APIs where clients specify **exactly what data they need**. Unlike REST, which has fixed endpoints, GraphQL uses a **single endpoint** and reduces over-fetching and under-fetching.

### 13. ****How do you add GraphQL to a Django project?****

**Answer:**

1. Install: pip install graphene-django
2. Add to INSTALLED\_APPS: 'graphene\_django'
3. Define a schema:

python

CopyEdit

import graphene

from graphene\_django.types import DjangoObjectType

from .models import Book

class BookType(DjangoObjectType):

class Meta:

model = Book

class Query(graphene.ObjectType):

all\_books = graphene.List(BookType)

def resolve\_all\_books(root, info):

return Book.objects.all()

schema = graphene.Schema(query=Query)

1. Connect in urls.py:

python

CopyEdit

from graphene\_django.views import GraphQLView

urlpatterns += [

path("graphql/", GraphQLView.as\_view(graphiql=True)),

]

### 14. ****When should you choose GraphQL over REST in Django?****

**Answer:**

* When frontend needs flexible, nested data queries
* When minimizing API calls is important
* When working with mobile apps or slow networks  
  Use GraphQL for **complex data interactions**. REST is better for simpler, standard APIs.

**🐌 1. “How would you debug a slow Django app?”**

**Answer:**

To debug a slow Django app, I follow a systematic approach:

1. **Enable Profiling Tools:**
   * Use **Django Debug Toolbar** in development to check query count, execution time, and template rendering.
   * Use **Silk** or **cProfile** for more detailed performance metrics.
2. **Check for N+1 Queries:**
   * Look for excessive DB hits inside loops. Use select\_related and prefetch\_related to reduce DB queries.
3. **Database Query Analysis:**
   * Enable queryset.query or use queryset.explain() to see how Django builds SQL.
   * Add indexes on frequently filtered or joined fields.
4. **Caching:**
   * Cache expensive computations or API responses using cache\_page, cache.set(), or template fragment caching.
   * Use Redis or Memcached for caching backend.
5. **Use Pagination:**
   * Never load all records at once. Use Django’s built-in Paginator for views or DRF’s pagination classes.
6. **Asynchronous Tasks:**
   * Offload tasks like email, PDF generation, or third-party API calls to **Celery**.
7. **Template Optimization:**
   * Avoid unnecessary loops or filters in templates.
   * Use {% include %} wisely to avoid nested rendering cost.

**⚠️ 2. “What are common mistakes you’ve made with Django in the past and how did you solve them?”**

**Answer:**

Here are a few honest mistakes I’ve made early in my Django journey and how I fixed them:

**Mistake 1: Putting too much logic in views instead of models or forms**

* ❌ I used to write all business logic in views, which made them long and untestable.
* ✅ I refactored logic into model methods or form clean methods to follow the **“fat models, thin views”** principle.

**Mistake 2: Forgetting to use select\_related / prefetch\_related**

* ❌ I once had a report page that loaded 500+ related objects—this caused N+1 query issues.
* ✅ After enabling Django Debug Toolbar, I identified the issue and replaced queries with select\_related() and prefetch\_related().

**Mistake 3: Leaving DEBUG=True in production**

* ❌ Once I deployed with DEBUG=True by accident—it exposed sensitive error data.
* ✅ I learned to always use environment variables and CI/CD pipelines that **fail the build** if DEBUG=True.

**Mistake 4: Overwriting migrations manually**

* ❌ I used to delete and recreate migrations instead of using makemigrations properly.
* ✅ I started using proper version control and running migrations on a **clone DB** before production deployment.

**🔄 3. “If multiple users edit the same model at once, how would you avoid conflicts?”**

**Answer:**

To handle concurrent editing and avoid conflicts (also known as the **lost update problem**), I use a combination of strategies:

✅ **1. Optimistic Locking (Manual):**  
Add a version or last\_updated field (e.g., DateTimeField with auto\_now) to detect stale updates.

python

CopyEdit

def save(self, \*args, \*\*kwargs):

if self.pk:

orig = MyModel.objects.get(pk=self.pk)

if orig.updated\_at != self.updated\_at:

raise ValidationError("Data has been modified by another user.")

super().save(\*args, \*\*kwargs)

✅ **2. Django-Concurrency Library:**  
For bigger projects, use the django-concurrency package, which handles version fields and raises exceptions if an outdated object is saved.

✅ **3. Locking at Database Level (Pessimistic):**  
Use select\_for\_update() inside a transaction if you **must** lock rows during editing.

python

CopyEdit

with transaction.atomic():

obj = MyModel.objects.select\_for\_update().get(pk=pk)

obj.field = "updated"

obj.save()

✅ **4. Show Edit Warnings in UI:**  
Display the "last edited" time to users in the frontend or warn if data has changed since form load.