Django

1. **What is MVC?**

    - **MVC** (Model-View-Controller) is an architectural design pattern that divides an application into three primary components:
        - **Model**: Represents the data structure and business logic of the application. It is responsible for interacting with the database.
        - **View**: Represents the user interface elements and presentation logic. It displays data to the user.
        - **Controller**: Handles user input, processes it, and updates the model and view. In the case of web applications, it would be responsible for interacting with the HTTP request and response.

    Django follows a **MTV** (Model-Template-View) pattern, which is very similar to MVC, but with a slight difference in terminology.

2. **Explain the architectural pattern of Django?**

    - Django follows the **MTV (Model-Template-View)** pattern, which is a variation of MVC.
        - **Model**: Defines the data structure and the database schema. It's usually represented as Django models, which define fields and their behavior.
        - **Template**: Defines how data is presented to the user. Django templates are used to display HTML and can include dynamic data from the model.
        - **View**: Handles the user request, processes it (usually interacting with the model), and returns a response (usually rendered via a template). In Django, views can be either functions or classes.
3. **What are the model inheritance styles in Django?**

    - Django supports three types of model inheritance:
        - **Abstract Base Class**: Defines common fields and behavior but does not create its own table. It is meant to be inherited by other models.
        - **Multi-table Inheritance**: Each model in the inheritance chain gets its own database table. Django automatically creates relationships between them (using foreign keys).
        - **Proxy Model**: A subclass of an existing model that doesn't create a new table. Instead, it allows you to add or modify methods for the model.
4. **What is WSGI and UWSGI?**

    - **WSGI** (Web Server Gateway Interface) is a standard interface between web servers and Python web applications. It defines how the web server communicates with the Python application to handle HTTP requests.
    - **uWSGI** is a tool that implements the WSGI standard and is commonly used in production to serve Django applications. It serves as a gateway between a web server (like Nginx) and the Python Django application.
5. **What is Mixins?**

    - **Mixins** are classes in Python used to add reusable pieces of functionality to other classes. In Django, mixins are commonly used to add common behavior to views or

serializers without writing repetitive code. For example, you might have a `LoginRequiredMixin` that ensures a user is logged in before accessing a view.

6. **Difference between class-based and function-based views?**

   - **Function-Based Views (FBVs)**: Simple Python functions that take an HTTP request as input and return an HTTP response. They are more explicit and suitable for small applications or simple logic.
   - **Class-Based Views (CBVs)**: More flexible and reusable, CBVs allow you to organize code into classes with methods that handle different HTTP methods (GET, POST, etc.). CBVs enable inheritance and mixins to easily extend functionality.

7. **Can we write a custom queryset in Django? How?**

   - Yes, you can write a custom queryset in Django by defining a custom manager. A manager is a class that handles query operations for your models. You can subclass `models.Manager` and add custom methods that return querysets. For example:

     ```
     python
     CopyEdit
     class CustomManager(models.Manager):
         def active(self):
             return self.filter(is_active=True)

     class Product(models.Model):
         name = models.CharField(max_length=100)
         is_active = models.BooleanField(default=True)
         objects = CustomManager()
     ```

8. **What is Model Manager?**

   - A **Model Manager** in Django is a class that is responsible for managing database queries for a model. By default, every model has a manager (usually named `objects`), but you can create custom managers to encapsulate commonly used queries. A custom manager can be used to add query methods that return specific subsets of data.

9. **Q Lookup/Operator**

   - Django's **Q objects** allow for more complex queries involving logical `AND`, `OR`, and negation. You can use `Q` objects to create filters with complex conditions. For example:

     ```
     python
     CopyEdit
     from django.db.models import Q
     result = MyModel.objects.filter(Q(field1='value1') |
     Q(field2='value2'))
     ```

10. **F Expression**

    - **F expressions** allow referencing model field values in queries, enabling operations to be performed directly in the database. This avoids the need to pull data into memory for updates. For example:

      ```
      python
      CopyEdit
      from django.db.models import F
      ```

```
Product.objects.filter(price__gt=F('discount_price'))
```

11. **Multiple Database Setup**
    - Django supports the use of multiple databases through database routers. You define database configurations in the `DATABASES` setting, and Django uses routers to determine which database to use for certain operations. This can be useful for scaling applications or separating different types of data (e.g., read vs. write databases).
12. **Django Connection Pooling (PostgreSQL)**
    - **Connection pooling** is a technique used to reuse database connections, improving performance by reducing the overhead of repeatedly opening and closing connections. For PostgreSQL, you can use third-party tools like `pgbouncer` or `django-db-pool` for connection pooling.
13. **Atomic Transactions in Django**
    - **Atomic transactions** ensure that database operations are executed as a single unit, meaning either all operations succeed, or none of them are applied. You can use the `@transaction.atomic` decorator or `with transaction.atomic():` block to handle this. This ensures that the database state is consistent.
14. **Difference between select_related and prefetch_related**
    - **select_related**: Works for foreign key and one-to-one relationships. It performs a SQL JOIN and retrieves the related objects in a single query.
    - **prefetch_related**: Works for many-to-many and reverse foreign key relationships. It performs a separate query for each related object and then combines them in Python.
15. **Explain Django Migrations, migrate vs makemigrations**
    - **Migrations** in Django are used to manage changes to the database schema.
        - `makemigrations` generates migration files based on changes to models.
        - `migrate` applies those migration files to the database to update its schema.
16. **How can we revert applied migration?**
    - You can revert an applied migration using the command `python manage.py migrate <app_name> <migration_name>`. For example, to undo all migrations for an app, use `python manage.py migrate app_name zero`.
17. **How can we create a custom authentication backend in Django?**
    - You can create a custom authentication backend by subclassing `django.contrib.auth.backends.BaseBackend`. Implement the `authenticate` method to define your authentication logic. For example:

```
python
CopyEdit
from django.contrib.auth.backends import BaseBackend
from django.contrib.auth.models import User

class CustomBackend(BaseBackend):
    def authenticate(self, request, username=None, password=None):
        try:
            user = User.objects.get(username=username)
            if user.check_password(password):
                return user
```

```
    except User.DoesNotExist:
        return None
```

18. **Explaining Caching**

- **Caching** in Django is a technique used to store expensive calculations or frequently accessed data to reduce response times and load on the database. Django provides various caching backends such as **Memcached**, **Redis**, and **database caching**. You can use `cache.set()` and `cache.get()` to store and retrieve data.

19. **Explaining Celery Integration**

- **Celery** is a task queue that allows you to run tasks asynchronously in the background. It is integrated with Django to handle tasks like sending emails or processing large datasets. Celery uses a message broker like **RabbitMQ** or **Redis** to manage tasks and queues.

20. **How to Secure Django App Before Production Mode?**

- Secure the Django app before production by:
  - Setting `DEBUG = False`.
  - Enabling HTTPS (SSL).
  - Using strong, unique secret keys.
  - Configuring proper database settings.
  - Enabling security headers like **Content Security Policy (CSP)**, **Strict-Transport-Security (STS)**, etc.
  - Implementing proper authentication and permissions.
  - Keeping Django and all dependencies updated.

21. **Logging HTTP Requests and Responses in Django with DRF**

- You can log HTTP requests and responses in Django using logging settings in `settings.py`. For more advanced logging, you can use middleware or packages like `django-request-logging` to log request/response data.

22. **Optimizing Django's QuerySet Performance**

- To optimize queries:
  - Use `select_related` and `prefetch_related` to reduce the number of queries.
  - Use **queryset caching** to avoid unnecessary database hits.
  - Avoid N+1 queries by using the appropriate relation-fetching methods.
  - Add indexes on frequently queried fields.

23. **What are Middlewares?**

- **Middleware** is a framework that processes requests and responses globally before or after they reach the view. It is useful for tasks such as logging, session management, or authentication. Django comes with built-in middlewares like `AuthenticationMiddleware`, `SessionMiddleware`, etc.

24. **What is Session Framework?**

- Django's **Session Framework** is used to store user-specific data across requests. This data is stored on the server, and only a session ID is sent to the client. It's commonly used for tracking user login sessions.

25. **Explain Django Reusability Code with Other Codes**

- Django promotes reusability by using **apps** and **modular code**. An app is a self-contained module with its models, views, and templates. You can reuse apps across different Django projects.

26. **How to Use Join in ORM?**

- Use **select_related** and **prefetch_related** for performing joins in Django ORM. `select_related` works for foreign key and one-to-one relationships, while `prefetch_related` works for many-to-many and reverse foreign key relationships.

27. **Write an ORM Query to Find Second Highest Salary of Employee Working in Python Department**

```
python
CopyEdit
from django.db.models import Max
max_salary =
Employee.objects.filter(department='Python').aggregate(Max('salary'))
['salary__max']
second_highest_salary = Employee.objects.filter(department='Python',
salary__lt=max_salary).aggregate(Max('salary'))['salary__max']
```

28. **What is Payload?**

- **Payload** refers to the actual data or body content in an HTTP request or response, typically sent in a POST or PUT request. It is the part of the message that carries the data.

29. **Why Use Management Command in Django?**

- Django management commands allow you to create custom commands that can be executed via `python manage.py <command_name>`. These are useful for running periodic tasks or performing administrative operations like data migration or batch processing.

30. **How to Use Pagination in Django?**

- Django provides **Paginator** for paginating querysets. You can use it as follows:

```
python
CopyEdit
from django.core.paginator import Paginator
paginator = Paginator(queryset, 10)  # Show 10 items per page
page_number = request.GET.get('page')
page_obj = paginator.get_page(page_number)
```

31. **What is Use of BaseModel in Django?**

- A **BaseModel** is an abstract model used to define common fields or methods that should be shared across multiple models. For example, a base model with `created_at` and `updated_at` timestamps can be used in multiple other models.

32. **Annotation vs Aggregation in Django**

- **Annotation** is used to add extra information to each object in a queryset, such as the count or sum of related objects.

- **Aggregation** calculates summary information (e.g., total, average) from a queryset and returns a single result.

33. **QuerySet Evaluation**

- Querysets are **lazy**, meaning they are not evaluated until needed. They are evaluated when you iterate over them, convert them to a list, or explicitly call methods like `count()`, `exists()`, or `get()`.

34. **Difference Between filter() and get() in Django ORM?**

- `filter()` returns a queryset that contains zero or more results.
- `get()` returns a single object or raises a `DoesNotExist` exception if no object is found.

35. **How Do You Update a Record in a Database Using Django ORM?**

- You can update a record by modifying its fields and calling `save()` on the object:

```
python
CopyEdit
obj = MyModel.objects.get(id=1)
obj.field = 'new_value'
obj.save()
```

36. **N + 1 Query Problem**

- The **N+1 query problem** occurs when querying related objects without optimizing for database hits, leading to an excessive number of queries. You can avoid this by using `select_related` or `prefetch_related` to fetch related objects in fewer queries.

Django Template Questions and Answers

1. What is Django Template?

Answer: A **Django template** is a text file that defines the structure or layout of a web page. It contains placeholders for dynamic data that is rendered when the page is viewed in a browser. Templates allow you to separate the presentation logic from the Python code (views), ensuring a cleaner and more maintainable codebase.

Django templates use a combination of HTML and template language to embed dynamic content.

2. How do you pass data to templates in Django?

Answer: Data is passed to templates using the context dictionary, which contains key-value pairs. The view function will pass the context to the template using the `render()` function.

Example:

```
python
CopyEdit
from django.shortcuts import render

def my_view(request):
    context = {'name': 'John'}
    return render(request, 'my_template.html', context)
```

In the template:

```
html
CopyEdit
<h1>Hello, {{ name }}!</h1>
```

3. What are template tags in Django?

Answer: Template tags are special constructs within the template language enclosed by `{% %}` that perform logic or actions such as loops, conditionals, or variable assignment.

Example:

- **For loop**:

  ```
  html
  CopyEdit
  <ul>
      {% for item in items %}
          <li>{{ item }}</li>
      {% endfor %}
  </ul>
  ```

- **If statement**:

  ```
  html
  CopyEdit
  {% if user.is_authenticated %}
      <p>Welcome back, {{ user.username }}!</p>
  {% else %}
      <p>Please log in to continue.</p>
  {% endif %}
  ```

4. What are template filters in Django?

Answer: Template filters are used to modify variables for display. They are applied using a pipe `|` symbol.

Example:

- **Uppercase filter**:

  ```
  html
  CopyEdit
  <p>{{ name|upper }}</p>  <!-- Converts the name to uppercase -->
  ```

- **Date filter**:

  ```
  html
  CopyEdit
  <p>{{ created_at|date:"Y-m-d" }}</p>  <!-- Formats the date -->
  ```

5. What are the most common template tags in Django?

Answer: Some of the most commonly used template tags are:

- **`{% for %}`**: Used for looping through a list or queryset.
- **`{% if %}`**: Used for conditional statements.
- **`{% include %}`**: Used to include another template.

- **`{% block %}`**: Used for defining blocks that can be overridden in child templates (used in template inheritance).
- **`{% extends %}`**: Used to extend a base template.
- **`{% url %}`**: Used to reverse a URL pattern and generate its corresponding URL.
- **`{% csrf_token %}`**: Generates a CSRF token for use in forms.

Example:

```
html
CopyEdit
{% for item in items %}
    <p>{{ item }}</p>
{% endfor %}
```

6. How do you inherit templates in Django?

Answer: Template inheritance allows you to define a common structure in a base template and extend it in child templates. This helps reduce redundancy and allows for better maintainability.

- **Base Template (`base.html`)**:

  ```
  html
  CopyEdit
  <html>
  <body>
      <header>
          <h1>My Website</h1>
      </header>
      <div>
          {% block content %}
          {% endblock %}
      </div>
  </body>
  </html>
  ```

- **Child Template (`home.html`)**:

  ```
  html
  CopyEdit
  {% extends 'base.html' %}

  {% block content %}
      <h2>Welcome to the Home Page!</h2>
  {% endblock %}
  ```

The `{% extends 'base.html' %}` tag tells Django to use `base.html` as the template layout, and the `{% block content %}` defines a section that can be overridden in the child template.

7. What is the `{% include %}` tag used for?

Answer: The `{% include %}` tag is used to include the content of another template into the current template. This is helpful for reusing common parts like headers, footers, or navigation bars.

Example:

```
html
CopyEdit
```

```
<!-- main.html -->
<html>
<body>
    <header>
        {% include 'header.html' %}
    </header>
    <div>
        <p>Content goes here.</p>
    </div>
</body>
</html>
```

In the above example, the content of `header.html` will be included wherever the `{% include 'header.html' %}` tag is placed.

8. What is the `static` tag in Django templates?

Answer: The `{% static %}` tag is used to link to static files like CSS, JavaScript, or image files in a template.

Example:

```
html
CopyEdit
<link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">
<img src="{% static 'images/logo.png' %}" alt="Logo">
```

The `{% static %}` tag will generate the correct URL for the static file, taking into account the `STATIC_URL` setting in the `settings.py` file.

9. What is the difference between `{{ }}` and `{% %}` in Django templates?

Answer:

- **`{{ }}`**: Used to output variables or expressions in a template. It evaluates and displays the value of a variable. Example:

  ```
  html
  CopyEdit
  <p>Hello, {{ user.username }}</p>
  ```

- **`{% %}`**: Used for template tags that perform logic or actions like loops, conditionals, including other templates, etc. Example:

  ```
  html
  CopyEdit
  {% for item in items %}
      <p>{{ item }}</p>
  {% endfor %}
  ```

10. How can you use Django templates with forms?

Answer: Django provides a way to easily render forms using templates. You can render form fields and their corresponding validation errors using Django's form system.

Example:

```
html
CopyEdit
```

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
```

- `{{ form.as_p }}` renders the form fields as `<p>` tags, but you can also use `{{ form.as_table }}` or manually render each field.

In the view:

```
python
CopyEdit
from django import forms
from django.shortcuts import render

class MyForm(forms.Form):
    name = forms.CharField()

def my_view(request):
    form = MyForm()
    return render(request, 'my_template.html', {'form': form})
```

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications, especially web services. It is based on a set of principles and constraints that, when followed, can help create scalable, efficient, and easy-to-maintain APIs. REST is widely used in web development and defines how clients and servers should communicate over HTTP.

Key Principles of REST:

1. **Statelessness**:
    - In REST, each request from the client to the server must contain all the information necessary to understand and process the request. The server does not store any information about the state of the client between requests. Each request is independent, and the server cannot rely on any previous interactions.
2. **Client-Server Architecture**:
    - REST uses a client-server model, where the client (such as a web browser or mobile app) and the server (which hosts the data) are separate entities. The client handles the user interface and user experience, while the server is responsible for processing requests and returning data.
3. **Uniform Interface**:
    - REST APIs have a uniform and consistent interface that simplifies interaction between clients and servers. This is achieved by using standard HTTP methods (GET, POST, PUT, DELETE) and standard formats like JSON or XML for data representation.

4. **Stateless Communication**:
    - Each interaction between the client and the server is independent and does not rely on previous requests. This makes the system more scalable because servers don't need to store information about each client between requests.
5. **Cacheable**:
    - Responses from the server can be explicitly marked as cacheable or non-cacheable. If a response is cacheable, the client can reuse the response for subsequent requests, improving performance and reducing the load on the server.
6. **Layered System**:
    - A REST API can be composed of multiple layers, such as load balancers, caching layers, or security layers. Each layer operates independently and can be changed without affecting the overall system.
7. **Code on Demand (Optional)**:
    - Servers can extend functionality by providing executable code (e.g., JavaScript) that clients can execute. This is an optional feature in REST and not widely used.

RESTful API Components:

- **Resources**: In REST, data or objects are considered resources, which can be identified using URLs (Uniform Resource Locators). For example, a URL like `https://api.example.com/users` refers to the collection of users.
- **HTTP Methods**:
    - **GET**: Used to retrieve data from the server.
    - **POST**: Used to create new resources on the server.
    - **PUT**: Used to update an existing resource or create a new one (if it doesn’t exist).
    - **DELETE**: Used to delete a resource on the server.
- **Representations**: Resources are represented in various formats, with JSON being the most common in REST APIs. When you request a resource, the server responds with the resource’s representation in a format the client can understand.

Example of a REST API Request and Response:

Let's say you have an API for managing users.

1. **Request (GET)**:
    - To get a list of all users:

      ```
      bash
      CopyEdit
      GET /users
      ```

    - The server responds with a JSON array representing the users:

      ```
      json
      CopyEdit
      [
        {
          "id": 1,
      ```

```
    "name": "John Doe",
    "email": "johndoe@example.com"
  },
  {
    "id": 2,
    "name": "Jane Smith",
    "email": "janesmith@example.com"
  }
]
```

2. **Request (POST)**:

   - To create a new user:

     ```
     bash
     CopyEdit
     POST /users
     Content-Type: application/json
     {
       "name": "Alice",
       "email": "alice@example.com"
     }
     ```

   - The server responds with a 201 status and the newly created user's data:

     ```
     json
     CopyEdit
     {
       "id": 3,
       "name": "Alice",
       "email": "alice@example.com"
     }
     ```

Benefits of REST:

- **Simplicity**: REST APIs are simple to use and understand because they rely on standard HTTP methods and conventions.
- **Scalability**: REST APIs can easily scale because they follow a stateless architecture, allowing for efficient handling of large amounts of traffic.
- **Flexibility**: REST allows clients and servers to evolve independently as long as they follow the basic principles of REST.
- **Interoperability**: REST APIs can be consumed by a variety of clients (web browsers, mobile apps, etc.) and work across different programming languages.

Conclusion:

REST is a powerful and widely-adopted architectural style that enables communication between clients and servers in a stateless, scalable, and flexible manner. It's commonly used to build web APIs due to its simplicity and reliance on standard protocols like HTTP.

DRF/REST API

1. **API Versioning in Django REST Framework**

- **API versioning** allows you to manage multiple versions of an API over time. In DRF, you can use different versioning schemes like **URLPathVersioning**, **NamespaceVersioning**, or **AcceptHeaderVersioning**.

2. **What is Serializer in Django REST Framework?**
    - A **serializer** is responsible for converting complex data types like querysets and model instances into native Python datatypes (such as dictionaries), which can then be rendered into JSON or XML for API responses.
3. **DRF vs Django**
    - **Django** is a full-stack web framework, while **DRF (Django REST Framework)** is a powerful toolkit specifically for building APIs. DRF provides features like serializers, viewsets, and routers to simplify API development.
4. **What is ViewSet in DRF?**
    - A **ViewSet** in DRF is a class-based view that provides methods like `.list()`, `.create()`, `.retrieve()`, `.update()`, and `.destroy()`. It allows for handling CRUD operations on your models with minimal code.
5. **Difference Between ViewSet and APIView**
    - **APIView** is a more general class-based view, while **ViewSet** is a specialized subclass designed to handle CRUD operations automatically. ViewSets are more concise when working with standard API behaviors.
6. **DRF Authentication**
    - DRF supports multiple types of authentication, including **SessionAuthentication**, **TokenAuthentication**, and **BasicAuthentication**. You can also implement custom authentication schemes if needed.
7. **Permission Classes in DRF**
    - **Permissions** in DRF are used to control access to views based on conditions like whether a user is authenticated, whether they are an admin, or whether they have specific attributes. Common permissions include **IsAuthenticated**, **IsAdminUser**, and **IsAuthenticatedOrReadOnly**.
8. **DRF Throttling**
    - **Throttling** is a way to limit the number of API requests that a user or client can make within a certain time period. DRF provides default throttling classes like **AnonRateThrottle** and **UserRateThrottle**, and you can define custom throttling classes.
9. **FilterBackends in DRF**
    - **FilterBackends** in DRF allow for filtering querysets based on request parameters. You can use built-in filters like `OrderingFilter` or `SearchFilter`, or you can implement custom filter backends.
10. **How Does DRF Handle Pagination?**
    - DRF provides several pagination classes like **PageNumberPagination**, **LimitOffsetPagination**, and **CursorPagination** to manage large result sets. You can configure the default pagination style in your settings.

API Observability

API Observability refers to the ability to monitor, measure, and understand the performance and health of an API. It helps teams track how the API is behaving in real-time, diagnose issues, and improve overall performance. Observability encompasses four pillars: **Metrics**, **Events**, **Logs**, and **Traces**. These pillars work together to provide a comprehensive view of an API's functioning.

1. **Metrics**:
    - **Metrics** are quantitative data points that represent the performance or health of an API over time. Examples include response times, request counts, error rates, throughput, and latency. Metrics allow you to understand the overall performance and health of your API by tracking key indicators.
    - Example: A metric could track the number of successful vs. failed requests or the average time taken to process requests.
2. **Events**:
    - **Events** capture specific actions or changes that occur within the API. These might include user logins, data updates, or status changes. Events are discrete occurrences that provide insights into the behavior of the API and its interactions with users.
    - Example: An event could represent a user logging in or a data update in the database.
3. **Logs**:
    - **Logs** are textual records of activities that occur within the system, often detailing errors, status updates, or key actions taken by the system. Logs provide detailed, timestamped information that can be used for debugging, auditing, and understanding the lifecycle of requests.
    - Example: A log could capture an error that occurs when the API fails to process a request or a debug message that helps developers identify issues in the code.
4. **Traces**:
    - **Traces** provide a detailed, end-to-end view of a request's journey through the system, from when it is received by the API to when a response is sent. Tracing enables you to understand the full lifecycle of a request, pinpoint where bottlenecks or failures occur, and analyze how different components interact.
    - Example: A trace might show how a request to retrieve data passes through different services, APIs, or databases before returning a response to the client.

Django Coding Questions

Middleware

Middleware is a way to process requests globally before they reach the view or after the response has been processed. Middleware is typically used for tasks such as authentication, logging, request/response modification, and security enforcement.

1. **Custom Middleware for Authentication and Permissions**
    - You can create custom middleware that ensures users are authenticated and have the required permissions before accessing certain views. For example:

```
python
CopyEdit
from django.http import HttpResponseForbidden
from django.shortcuts import redirect

class AuthMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Check if the user is authenticated
        if not request.user.is_authenticated:
            return redirect('login')

        # Check for specific permissions (if needed)
        if not request.user.has_perm('app.view_data'):
            return HttpResponseForbidden('You do not have permission
to view this data.')

        # Process the request
        response = self.get_response(request)
        return response
```

2. **Middleware for Logging Requests and Responses**

- Logging can help with debugging and monitoring API requests. A middleware could log details about each request and its response:

```
python
CopyEdit
import logging
from time import time

logger = logging.getLogger(__name__)

class LoggingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        start_time = time()
        response = self.get_response(request)
        end_time = time()

        # Log request and response details
        logger.info(f"Request: {request.method} {request.path} -
Response: {response.status_code} - Time: {end_time - start_time}s")

        return response
```

3. **Tracking View Performance**

- You can track the performance of your views by measuring the time taken to process requests and responses:

```
python
CopyEdit
from time import time

class PerformanceTrackingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
```

```
    def __call__(self, request):
        start_time = time()
        response = self.get_response(request)
        end_time = time()

        # Calculate the time taken and log it
        execution_time = end_time - start_time
        print(f"View processing time: {execution_time}s")

        return response
```

4. **Blocking Requests from Specific IPs**

- You can block requests from specific IP addresses using middleware:

```
python
CopyEdit
class BlockIPMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        blocked_ips = ['192.168.1.1', '192.168.1.2']
        if request.META.get('REMOTE_ADDR') in blocked_ips:
            return HttpResponseForbidden("Your IP is blocked.")

        return self.get_response(request)
```

5. **Compressing Response Content**

- Middleware can compress response content to reduce the amount of data transferred:

```
python
CopyEdit
from django.http import HttpResponse
import gzip
from io import BytesIO

class CompressionMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)

        # Only compress if the response is text-based
        if response.get('Content-Type', '').startswith('text'):
            buffer = BytesIO()
            with gzip.GzipFile(fileobj=buffer, mode='w') as f:
                f.write(response.content)
            response.content = buffer.getvalue()
            response['Content-Encoding'] = 'gzip'

        return response
```

6. **Modifying HTTP Headers**

- You can modify headers for requests or responses:

```
python
CopyEdit
class HeaderModificationMiddleware:
```

```
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)
        # Add custom headers
        response['X-Custom-Header'] = 'MyCustomValue'
        return response
```

7. **Rate Limiting Requests**

- You can limit the number of requests a user can make in a given time period:

```
python
CopyEdit
from time import time

class RateLimitMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        self.user_requests = {}

    def __call__(self, request):
        user_ip = request.META.get('REMOTE_ADDR')
        current_time = time()

        if user_ip in self.user_requests:
            request_count, last_request_time =
self.user_requests[user_ip]
            if current_time - last_request_time < 60:  # Rate limit
to 1 request per minute
                if request_count >= 5:
                    return HttpResponseForbidden("Rate limit
exceeded.")
            self.user_requests[user_ip] = (request_count + 1,
current_time)
        else:
            self.user_requests[user_ip] = (1, current_time)

        return self.get_response(request)
```

8. **Enforcing HTTPS**

- Middleware can enforce HTTPS:

```
python
CopyEdit
class HTTPSMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        if not request.is_secure():
            return
HttpResponsePermanentRedirect(f'https://{request.get_host()}
{request.get_full_path()}')
        return self.get_response(request)
```

9. **API Versioning Middleware**

- Middleware can handle versioning by routing requests to the appropriate API version:

```
python
CopyEdit
class APIVersionMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        api_version = request.META.get('HTTP_X_API_VERSION', 'v1')
        request.version = api_version
        return self.get_response(request)
```

10. **Custom Middleware for Logging Time Taken**

- You can create a custom middleware that logs the time taken to process each request:

```
python
CopyEdit
from time import time

class LogRequestTimeMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        start_time = time()
        response = self.get_response(request)
        end_time = time()

        # Log the time taken for the request
        print(f"Request processed in {end_time - start_time:.2f}
seconds.")
        return response
```

Django Coding Question: Top 5 Authors with Most Books

To find the top 5 authors with the most books, assuming you have an `Author` model and a `Book` model where each book is linked to an author:

```
python
CopyEdit
from django.db.models import Count

# Query to find the top 5 authors with the most books
top_authors = Author.objects.annotate(book_count=Count('book')).order_by('-
book_count')[:5]

# Display the authors and their book count
for author in top_authors:
    print(f"{author.name}: {author.book_count} books")
```

REST API

Custom Validation in Serializer

To ensure that the title of a blog post is unique, you can add a custom validation method in the serializer:

```
python
CopyEdit
```

```
from rest_framework import serializers
from .models import BlogPost

class BlogPostSerializer(serializers.ModelSerializer):
    class Meta:
        model = BlogPost
        fields = ['title', 'content']

    def validate_title(self, value):
        if BlogPost.objects.filter(title=value).exists():
            raise serializers.ValidationError("Title must be unique.")
        return value
```

ATM Withdrawal API

For an API that handles ATM withdrawals with specific requirements (6-digit pin, amounts in multiples of 100 and less than 1000), you could create a serializer and view like this:

```
python
CopyEdit
from rest_framework import serializers, viewsets
from rest_framework.response import Response
from rest_framework.decorators import action
from django.core.exceptions import ValidationError

class ATMWithdrawalSerializer(serializers.Serializer):
    atm_pin = serializers.CharField(max_length=6, min_length=6)
    amount = serializers.IntegerField()

    def validate_atm_pin(self, value):
        if not value.isdigit():
            raise ValidationError("ATM PIN must be numeric.")
        return value

    def validate_amount(self, value):
        if value % 100 != 0 or value > 1000:
            raise ValidationError("Amount must be a multiple of 100 and less
than 1000.")
        return value

class ATMWithdrawalViewSet(viewsets.ViewSet):
    @action(detail=False, methods=['post'])
```