Django Developers

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain Django's request-response cycle in detail, including middleware execution order.

Request-Response Cycle Overview

Django's request-response cycle involves multiple layers of processing:

- **Request Phase:** WSGI server receives HTTP request and creates HttpRequest object
- **Middleware (Request):** Executes process_request() methods top-to-bottom
- **URL Resolution:** URLconf matches request path to view function
- **Middleware (View):** Executes process_view() methods top-to-bottom
- **View Execution:** View function processes request and returns HttpResponse
- **Middleware (Response):** Executes process_response() methods bottom-to-top
- **Response Delivery:** WSGI server sends response to client

Exception Handling: If an exception occurs, process_exception() middleware methods execute bottom-to-top before returning error response.

Critical Detail: Response middleware always executes in reverse order, ensuring proper cleanup and wrapping of responses. Template response middleware (process_template_response) executes before process_response for TemplateResponse objects.

2. How does Django's select_related() differ from prefetch_related()? When would you use each?

Query Optimization Strategies

select_related():

- Uses SQL JOIN to fetch related objects in single query
- Works with ForeignKey and OneToOneField relationships
- Returns QuerySet with related objects cached
- Best for single-valued relationships

```
articles = Article.objects.select_related('author', 'category')
# Generates single JOIN query
```

prefetch_related():

- Performs separate query for each relationship
- Works with ManyToManyField, reverse ForeignKey, and GenericRelation
- Does Python-side joining of results
- Best for multi-valued relationships

```
authors = Author.objects.prefetch_related('articles')
# Generates two queries: one for authors, one for articles
```

Performance Consideration: select_related() is more efficient for one-to-one/many-to-one, while prefetch_related() prevents cartesian product explosion in many-to-many scenarios. Combine both for complex relationship graphs.

3. What are Django signals and what are their potential pitfalls in production systems?

Django Signals Architecture

Signals provide decoupled communication between Django components through the observer pattern. Common built-in signals include pre_save, post_save, pre_delete, post_delete, m2m_changed, and request_started.

Basic Usage:

```
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

Production Pitfalls:

- **Performance:** Signals execute synchronously, blocking the request-response cycle
- **Transaction Safety:** post_save fires before transaction commit; use transaction.on_commit() for external API calls
- **Hidden Dependencies:** Signal handlers create implicit coupling that's hard to trace
- **Testing Complexity:** Signals fire during tests unless explicitly disconnected
- **Recursive Triggers:** Signal handlers that modify models can cause infinite loops

Best Practice: Use signals sparingly; prefer explicit method calls or consider task queues (Celery) for time-consuming operations.

4. Explain Django's database transaction management and the difference between ATOMIC_REQUESTS and atomic() decorator.

Transaction Management Strategies

ATOMIC_REQUESTS Setting:

```
DATABASES = {
    'default': {
        'ATOMIC_REQUESTS': True
    }
}
```

Wraps each view in a transaction. Commits on success, rolls back on exceptions. Simple but inflexible—entire view is atomic.

atomic() Decorator/Context Manager:

```
from django.db import transaction

@transaction.atomic
def create_order(user, items):
    order = Order.objects.create(user=user)
    for item in items:
        OrderItem.objects.create(order=order, item=item)
```

Key Differences:

- **Granularity:** atomic() allows fine-grained control over transaction boundaries
- **Nesting:** atomic() blocks can be nested; inner blocks create savepoints
- **Performance:** ATOMIC_REQUESTS adds overhead to all views, even read-only ones
- **Exception Handling:** atomic() allows catching exceptions outside transaction scope

Advanced Pattern: Use transaction.on_commit() to defer actions (sending emails, cache invalidation) until transaction succeeds, preventing premature side effects.

5. How would you implement custom model managers and querysets for reusable filtering logic?

Custom Managers and QuerySets

Custom QuerySet Pattern:

```
class PublishedQuerySet(models.QuerySet):
    def published(self):
        return self.filter(status='published', publish_date__lte=timezone.now())
    def featured(self):
        return self.filter(is_featured=True)

class ArticleManager(models.Manager):
    def get_queryset(self):
        return PublishedQuerySet(self.model, using=self._db)

    def published(self):
        return self.get_queryset().published()
```

Model Implementation:

```
class Article(models.Model):
    objects = ArticleManager()
    # Usage: Article.objects.published().featured()
```

Advantages:

- **Chainability:** QuerySet methods can be chained together
- **Reusability:** Encapsulates complex filtering logic
- **DRY Principle:** Eliminates repeated filter conditions across codebase
- **Type Safety:** IDE autocomplete works with custom methods

Best Practice: Use QuerySet.as_manager() shortcut for simple cases: objects = PublishedQuerySet.as_manager()

6. What are the security implications of Django's ORM and how do you prevent SQL injection and mass assignment vulnerabilities?

ORM Security Considerations

SQL Injection Prevention:

Django's ORM automatically parameterizes queries, but raw SQL requires caution:

```
# SAFE: Parameterized query
User.objects.raw('SELECT * FROM users WHERE id = %s', [user_id])

# UNSAFE: String interpolation
User.objects.raw('SELECT * FROM users WHERE id = %s' % user_id)

# SAFE: Using extra() with params
User.objects.extra(where=['age > %s'], params=[18])
```

Mass Assignment Protection:

Django doesn't have Rails-style mass assignment by default, but form handling requires care:

```
# VULNERABLE: Directly using request data
user = User.objects.create(**request.POST.dict())

# SECURE: Use forms for validation
form = UserForm(request.POST)
if form.is_valid():
    user = form.save()
```

Additional Security Measures:

- **Field-level permissions:** Override save() to validate field changes
- **Read-only fields:** Define in ModelForm.Meta.fields or exclude
- **Query parameterization:** Always use params argument in raw queries
- **Avoid extra():** Prefer annotate() and filter() for complex queries

7. Explain Django's caching framework and describe strategies for implementing multi-layer caching in a high-traffic application.

Multi-Layer Caching Strategy

Cache Backend Configuration:

```
CACHES = {
    'default': {'BACKEND': 'django.core.cache.backends.redis.RedisCache',
                'LOCATION': 'redis://127.0.0.1:6379/1'},
    'sessions': {'BACKEND': 'django.core.cache.backends.redis.RedisCache',
                 'LOCATION': 'redis://127.0.0.1:6379/2'}
}
```

Caching Layers:

- **Per-Site Cache:** Middleware caches entire site (CACHE_MIDDLEWARE_SECONDS)
- **Per-View Cache:** @cache_page(60 * 15) decorator for specific views
- **Template Fragment Cache:** {% cache 500 sidebar %} for expensive template sections
- **Low-Level Cache:** cache.set('key', value, timeout) for granular control

Advanced Pattern:

```
from django.core.cache import cache
from django.views.decorators.cache import cache_page

def get_user_data(user_id):
    cache_key = f'user_data_{user_id}'
    data = cache.get(cache_key)
    if data is None:
        data = expensive_query(user_id)
        cache.set(cache_key, data, 3600)
    return data
```

Invalidation Strategy: Use signals or post_save hooks to invalidate specific keys. Consider cache versioning for atomic updates.

8. How do Django's async views work and what are the considerations when mixing sync and async code?

Asynchronous Django Views

Async View Definition:

```
from django.http import JsonResponse
import asyncio

async def async_view(request):
    data = await fetch_external_api()
    await asyncio.sleep(1)
    return JsonResponse({'data': data})

# ASGI required in settings
ASGI_APPLICATION = 'myproject.asgi.application'
```

Sync/Async Boundary Management:

```
from asgiref.sync import sync_to_async, async_to_sync

# Call sync ORM from async view
async def async_view(request):
    users = await sync_to_async(list)(User.objects.all())
    return JsonResponse({'count': len(users)})

# Call async function from sync view
def sync_view(request):
    result = async_to_sync(async_function)()
```

Critical Considerations:

- **ORM Limitations:** Django ORM is synchronous; wrap queries in sync_to_async()
- **Thread Safety:** Database connections aren't thread-safe in async context
- **Middleware Compatibility:** Some middleware may not support async
- **Performance:** Async shines for I/O-bound operations, not CPU-bound tasks
- **Deployment:** Requires ASGI server (Uvicorn, Daphne) instead of WSGI

9. Describe Django's middleware architecture and how to implement custom middleware

for request/response modification.

Custom Middleware Implementation

Modern Middleware Class (Django 1.10+):

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

    def __call__(self, request):
        start_time = time.time()
        response = self.get_response(request)
        duration = time.time() - start_time
        response['X-Request-Duration'] = str(duration)
        return response

    def process_exception(self, request, exception):
        logger.error(f'Exception: {exception}')
        return None
```

Middleware Hooks:

- **__call__:** Main request/response processing
- **process_view:** Called before view execution, receives view function and args
- **process_exception:** Called when view raises exception
- **process_template_response:** Called for TemplateResponse objects

Configuration:

```
MIDDLEWARE = [
    'myapp.middleware.TimingMiddleware',
    'django.middleware.security.SecurityMiddleware',
]
```

Use Cases: Authentication, request logging, rate limiting, response compression, custom headers, A/B testing, feature flags.

10. What strategies would you use to optimize Django ORM queries for a complex reporting dashboard with multiple aggregations and joins?

Query Optimization Strategies

Aggregation and Annotation:

```
from django.db.models import Count, Avg, Prefetch, Q, F

reports = Order.objects.select_related('customer').annotate(
    total_items=Count('items'),
    avg_price=Avg('items__price'),
    revenue=Sum(F('items__price') * F('items__quantity'))
).filter(created_at__gte=start_date)
```

Advanced Optimization Techniques:

- **Prefetch Objects:** Customize prefetch queries with Prefetch() for filtered relations
- **only()/defer():** Load specific fields to reduce data transfer
- **iterator():** Stream large querysets without caching all results
- **Raw SQL:** Use raw() or execute() for complex queries ORM can't optimize
- **Database Views:** Create materialized views for expensive aggregations

```
# Custom prefetch with filtering
authors = Author.objects.prefetch_related(
    Prefetch('articles',
             queryset=Article.objects.filter(published=True),
             to_attr='published_articles')
)
```

Monitoring: Use django-debug-toolbar, connection.queries, or explain() to analyze query plans. Consider database-level optimizations: indexes, query caching, read replicas.

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) cache in Python with O(1) time complexity for both get and put operations.

LRU Cache Implementation

An **LRU cache** can be implemented using a combination of a **doubly linked list** and a **hash map (dictionary)**. The hash map provides O(1) access to cache items, while the doubly linked list maintains the order of usage.

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return -1
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)
```

Time Complexity: O(1) for both get and put operations

Space Complexity: O(capacity)

2. How would you find all pairs in an array that sum to a target value? What is the time complexity?

Finding Pair Sum

Use a **hash set** to store elements as you iterate through the array. For each element, check if the complement (target - current element) exists in the set.

```
def find_pairs(arr, target):
    seen = set()
    pairs = []
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

Time Complexity: O(n) - single pass through the array

Space Complexity: O(n) - for the hash set

Example: find_pairs([2, 7, 11, 15], 9) returns [(2, 7)]

3. Implement a stack that supports push, pop, and retrieving the minimum element in constant time.

Min Stack Implementation

Maintain two stacks: one for all elements and another for tracking minimum values. Each time you push, also push the current minimum onto the min stack.

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        min_val = min(val, self.min_stack[-1] if self.min_stack else val)
        self.min_stack.append(min_val)

    def pop(self):
        self.stack.pop()
        self.min_stack.pop()

    def get_min(self):
        return self.min_stack[-1]
```

Time Complexity: O(1) for all operations

Space Complexity: O(n) for storing two stacks

4. Explain the difference between Python's list, tuple, set, and dictionary. When would you use each?

Python Data Structures Comparison

- **List:** Ordered, mutable sequence. Use for ordered collections that need modification. Time: O(1) append, O(n) insert/delete at beginning.
- **Tuple:** Ordered, immutable sequence. Use for fixed data, dictionary keys, or function returns. More memory efficient than lists.
- **Set:** Unordered, mutable collection of unique elements. Use for membership testing, removing duplicates, or set operations. Time: O(1) for add/remove/lookup.
- **Dictionary:** Unordered key-value pairs. Use for fast lookups by key, caching, or counting. Time: O(1) average for get/set operations.

Performance tip: Use sets for membership testing instead of lists when dealing with large collections.

5. Implement a sliding window algorithm to find the maximum sum of k consecutive elements in an array.

Sliding Window Maximum Sum

The **sliding window technique** avoids recalculating the entire sum by subtracting the element leaving the window and adding the new element entering it.

```
def max_sum_subarray(arr, k):
    if len(arr) < k:
        return None

    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, window_sum)

    return max_sum
```

Time Complexity: O(n) - single pass through array

Space Complexity: O(1) - constant extra space

Naive approach would be O(n*k) by recalculating sum for each window.

6. How would you detect a cycle in a linked list? Provide the algorithm and its complexity.

Floyd's Cycle Detection Algorithm

Use **two pointers** (slow and fast). Slow moves one step at a time, fast moves two steps. If there's a cycle, they will eventually meet.

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

Time Complexity: O(n) - at most n iterations

Space Complexity: O(1) - only two pointers used

This is also known as the **tortoise and hare algorithm**.

7. Implement a function to reverse a string in-place. What are the challenges in Python?

String Reversal in Python

Challenge: Strings in Python are **immutable**, so true in-place reversal isn't possible. You must convert to a mutable type like a list.

```
def reverse_string(s):
    return s[::-1]

def reverse_string_manual(s):
    chars = list(s)
    left, right = 0, len(chars) - 1
    while left < right:
        chars[left], chars[right] = chars[right], chars[left]
        left += 1
        right -= 1
    return ''.join(chars)
```

Time Complexity: O(n) for both approaches

Space Complexity: O(n) due to string immutability

The slicing approach [::-1] is more Pythonic and optimized at C level.

8. What is a hash collision and how does Python's dictionary handle it?

Hash Collisions in Python Dictionaries

A **hash collision** occurs when two different keys produce the same hash value. Python dictionaries handle collisions using **open addressing with random probing**.

How it works:

- When a collision occurs, Python probes for the next available slot using a pseudo-random sequence
- The probe sequence depends on the hash value and number of collisions
- Dictionary is resized when it becomes 2/3 full to maintain performance

- Average case: O(1) lookup, worst case: O(n)

Hash function requirements:

- Deterministic: same input always produces same hash
- Uniform distribution to minimize collisions
- Immutable keys only (strings, tuples, numbers)

9. Implement a binary search algorithm and explain when it's preferable over linear search.

Binary Search Implementation

Binary search efficiently finds an element in a **sorted array** by repeatedly dividing the search interval in half.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Time Complexity: O(log n) vs O(n) for linear search

When to use binary search:

- Array is sorted or can be sorted once and searched multiple times
- Large datasets where O(log n) provides significant advantage
- Random access is available (arrays, not linked lists)

10. Explain the time complexity of common operations on Python's list, set, and dictionary.

Python Data Structure Time Complexities

List Operations:

- Access by index: O(1)
- Append: O(1) amortized
- Insert/Delete at beginning: O(n)
- Search (in operator): O(n)
- Pop from end: O(1), from beginning: O(n)

Set Operations:

- Add: O(1) average
- Remove: O(1) average
- Membership test (in): O(1) average
- Union/Intersection: O(len(s1) + len(s2))

Dictionary Operations:

- Get/Set item: O(1) average
- Delete item: O(1) average
- Membership test (in): O(1) average
- Iteration: O(n)

Key insight: Use sets/dicts for fast lookups, lists for ordered sequences.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service using Django. What are the key architectural components and trade-offs?

Architecture Overview

A scalable URL shortener requires careful consideration of several components:

- **URL Generation Strategy:** Use base62 encoding with auto-incrementing IDs or hash-based approaches (MD5/SHA256 truncated). Base62 provides short URLs while avoiding ambiguous characters.
- **Database Design:** Store mappings in PostgreSQL with indexes on short_code. Consider partitioning by creation date for historical data.
- **Caching Layer:** Redis for hot URLs (80/20 rule - 20% URLs get 80% traffic). Use TTL-based eviction.
- **Rate Limiting:** Implement per-IP and per-user limits using Django middleware with Redis backend.

Key Design Decisions

- **ID Generation:** Use distributed ID generation (Snowflake-like) to avoid collisions in multi-server setup
- **Write vs Read Optimization:** Reads >> Writes, so optimize for read performance with aggressive caching
- **Analytics:** Async processing using Celery for click tracking to avoid blocking redirects

```
class URLMapping(models.Model):
    short_code = models.CharField(max_length=10, unique=True, db_index=True)
    original_url = models.URLField(max_length=2048)
    created_at = models.DateTimeField(auto_now_add=True)
    click_count = models.BigIntegerField(default=0)

    class Meta:
        indexes = [models.Index(fields=['created_at'])]
```

Trade-offs

- **Consistency vs Availability:** Favor availability (AP in CAP theorem) - temporary inconsistency in click counts is acceptable
- **Storage:** 1 billion URLs × 100 bytes = ~100GB manageable in single DB, but plan for sharding by hash ranges

2. How would you design a real-time notification system in Django that supports WebSockets, push notifications, and email with millions of users?

System Architecture

- **Django Channels:** Use Django Channels with Redis channel layer for WebSocket connections. Deploy separate ASGI servers for WebSocket handling.
- **Message Queue:** Celery with RabbitMQ/Redis for async processing of notifications
- **Notification Storage:** PostgreSQL for persistence with proper indexing on user_id and read status
- **Push Services:** Firebase Cloud Messaging (FCM) for mobile, Web Push API for browsers

Scalability Considerations

- **Connection Management:** Each ASGI worker handles ~10k concurrent WebSocket

connections. Use load balancer with sticky sessions.
- **Fan-out Problem:** For broadcasts to millions, use pub/sub pattern. Publish once to Redis, let workers fan out to connected clients.
- **Presence Service:** Track online users in Redis with TTL, update on heartbeat

```
class NotificationConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.user_id = self.scope['user'].id
        await self.channel_layer.group_add(
            f'user_{self.user_id}', self.channel_name
        )
        await self.accept()
```

Delivery Guarantees

- **At-least-once delivery:** Store notifications in DB first, mark as delivered after ACK
- **Fallback mechanism:** WebSocket → Push → Email with priority queue
- **Rate limiting:** Per-user notification throttling to prevent spam

3. Design a social media feed system in Django that handles millions of posts with personalized ranking. How do you handle the fan-out problem?

Feed Architecture Patterns

Two main approaches with different trade-offs:

- **Fan-out on Write (Push Model):** When user posts, write to all followers' feeds immediately. Good for read-heavy, but expensive writes for celebrities.
- **Fan-out on Read (Pull Model):** Aggregate feed on request from followed users' posts. Expensive reads, but cheaper writes.
- **Hybrid Approach:** Use push for normal users (<10k followers), pull for celebrities. Best of both worlds.

Database Schema

```
class Post(models.Model):
    author = models.ForeignKey(User)
    content = models.TextField()
    created_at = models.DateTimeField(db_index=True)

class Feed(models.Model):
    user = models.ForeignKey(User)
    post = models.ForeignKey(Post)
    created_at = models.DateTimeField(db_index=True)
```

Ranking & Personalization

- **Scoring Algorithm:** Combine recency, engagement (likes/comments), author relationship strength
- **ML Integration:** Use Celery to compute personalized scores asynchronously, cache top 100 posts per user in Redis
- **Real-time Updates:** WebSocket notifications for new posts, client polls for ranking updates

Scalability Solutions

- **Caching:** Redis sorted sets for feed storage, TTL of 24 hours
- **Database Sharding:** Shard Feed table by user_id hash
- **Pagination:** Cursor-based pagination using created_at + id composite key

4. How would you architect a multi-tenant SaaS application in Django with proper data isolation and performance optimization?

Multi-Tenancy Approaches

- **Shared Database, Shared Schema:** Add tenant_id to all tables. Simplest but requires careful query filtering.
- **Shared Database, Separate Schemas:** PostgreSQL schemas per tenant. Better isolation, moderate complexity.

- **Separate Databases:** Complete isolation, easier compliance, but higher operational overhead.

Recommended: Shared Schema with Row-Level Security

```
class TenantAwareModel(models.Model):
    tenant = models.ForeignKey(Tenant)

    class Meta:
        abstract = True

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

    def __call__(self, request):
        request.tenant = get_tenant_from_request(request)
        return self.get_response(request)
```

Data Isolation Strategies

- **Custom Manager:** Override default manager to auto-filter by tenant
- **Middleware:** Set tenant context at request level, use thread-local storage
- **Database Constraints:** Add CHECK constraints and RLS policies in PostgreSQL

Performance Optimization

- **Connection Pooling:** Use PgBouncer with separate pools per tenant for large tenants
- **Caching:** Namespace cache keys by tenant_id
- **Query Optimization:** Composite indexes on (tenant_id, frequently_queried_field)
- **Tenant Tiering:** Route premium tenants to dedicated resources

5. Design a distributed task processing system in Django for handling millions of background jobs with priority queues, retries, and monitoring.

Architecture Components

- **Task Queue:** Celery with Redis as broker and result backend. RabbitMQ for more complex routing needs.
- **Priority Queues:** Separate queues for high/medium/low priority tasks with dedicated workers
- **Task Storage:** PostgreSQL for task metadata, results, and audit trail
- **Monitoring:** Flower for Celery monitoring, custom metrics to Prometheus

Reliability Patterns

```
@shared_task(bind=True, max_retries=3)
def process_payment(self, order_id):
    try:
        order = Order.objects.select_for_update().get(id=order_id)
        # Idempotent processing
        if order.status == 'processed':
            return
        payment_gateway.charge(order)
        order.status = 'processed'
        order.save()
    except Exception as exc:
        raise self.retry(exc=exc, countdown=60 * (2 ** self.request.retries))
```

Scalability Design

- **Worker Scaling:** Auto-scale workers based on queue depth using Kubernetes HPA
- **Task Routing:** Route tasks to specialized workers (CPU-intensive, I/O-bound, GPU)
- **Rate Limiting:** Implement token bucket algorithm for API-calling tasks
- **Circuit Breaker:** Pause tasks when downstream services fail

Monitoring & Observability

- **Metrics:** Task success/failure rates, execution time percentiles, queue lengths

- **Dead Letter Queue:** Move failed tasks after max retries for manual inspection
- **Distributed Tracing:** Correlate tasks with originating requests using trace IDs

6. How would you design a file upload and processing system in Django that handles large files (GBs), virus scanning, format conversion, and CDN delivery?

Upload Strategy

- **Direct Upload to S3:** Generate pre-signed URLs, clients upload directly to S3, bypassing Django servers. Reduces server load significantly.
- **Chunked Upload:** For browser uploads, use chunked multipart upload with resumability
- **Webhook Callback:** S3 triggers webhook to Django after upload completes

Processing Pipeline

```
@receiver(post_save, sender=UploadedFile)
def process_file(sender, instance, created, **kwargs):
    if created:
        chain(
            scan_virus.s(instance.id),
            extract_metadata.s(),
            generate_thumbnails.s(),
            convert_format.s(),
            publish_to_cdn.s()
        ).apply_async()
```

Architecture Components

- **Storage:** S3 for raw uploads, separate bucket for processed files
- **Virus Scanning:** ClamAV in containerized workers, scan before processing
- **Format Conversion:** FFmpeg for video, ImageMagick for images, dedicated worker pools
- **CDN:** CloudFront with signed URLs for private content

Scalability & Performance

- **Async Processing:** All processing in Celery tasks, update status in DB
- **Worker Specialization:** Separate workers for CPU-intensive (conversion) vs I/O (scanning)
- **Progress Tracking:** WebSocket updates for upload progress, polling for processing status
- **Cleanup:** Scheduled tasks to remove temporary files and old versions

7. Design a rate limiting and API throttling system for a Django REST API serving millions of requests per day with different tiers of users.

Rate Limiting Strategies

- **Token Bucket Algorithm:** Allows bursts while maintaining average rate. Best for API throttling.
- **Fixed Window:** Simple but allows burst at window boundaries
- **Sliding Window Log:** Most accurate but memory intensive
- **Sliding Window Counter:** Good balance of accuracy and efficiency

Implementation Approach

```
class RateLimitMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        self.redis = redis.Redis()

    def __call__(self, request):
        key = f'rl:{request.user.id}:{request.path}'
        limit = self.get_user_limit(request.user)
        if not self.check_rate_limit(key, limit):
            return JsonResponse({'error': 'Rate limit exceeded'}, status=429)
```

Multi-Tier Design

- **User Tiers:** Free (100 req/hour), Basic (1000 req/hour), Premium (10000 req/hour), Enterprise

(unlimited)
- **Endpoint-Specific Limits:** Different limits for read vs write operations
- **IP-Based Fallback:** Rate limit by IP for unauthenticated requests
- **Distributed Rate Limiting:** Use Redis with Lua scripts for atomic operations across servers

Advanced Features

- **Rate Limit Headers:** Return X-RateLimit-Remaining, X-RateLimit-Reset in responses
- **Graceful Degradation:** Queue requests when near limit, process with delay
- **Dynamic Limits:** Adjust limits based on system load and user behavior
- **Bypass Mechanisms:** Whitelist internal services, admin override capability

8. How would you design a search system in Django with full-text search, filtering, faceting, and auto-suggestions for an e-commerce platform with millions of products?

Search Engine Choice

- **Elasticsearch:** Best for complex queries, faceting, and analytics. Horizontal scaling built-in.
- **PostgreSQL Full-Text Search:** Good for simpler use cases, no additional infrastructure
- **Recommendation:** Elasticsearch with django-elasticsearch-dsl for e-commerce scale

Index Design

```
from elasticsearch_dsl import Document, Text, Keyword, Integer

class ProductDocument(Document):
    name = Text(analyzer='standard', fields={'raw': Keyword()})
    description = Text(analyzer='english')
    category = Keyword()
    price = Integer()

    class Index:
        name = 'products'
        settings = {'number_of_shards': 3, 'number_of_replicas': 2}
```

Key Features Implementation

- **Full-Text Search:** Multi-field search with boosting (name^3, description^1). Use edge n-grams for partial matching.
- **Faceting:** Aggregations on category, price ranges, brand, ratings. Return counts for filter options.
- **Auto-Suggestions:** Completion suggester with prefix matching. Separate index for popular queries.
- **Relevance Tuning:** Combine text score with popularity, sales, and ratings using function_score query

Scalability & Performance

- **Index Synchronization:** Use Celery tasks triggered by Django signals to update Elasticsearch
- **Caching:** Cache popular searches in Redis with TTL
- **Query Optimization:** Use filters instead of queries when possible (cached), limit result window
- **Monitoring:** Track slow queries, index size, cluster health

9. Design a session management and authentication system for a Django application that needs to support SSO, OAuth2, JWT tokens, and maintain stateless architecture.

Authentication Architecture

- **JWT for Stateless Auth:** Access tokens (15 min) + Refresh tokens (7 days) stored in httpOnly cookies
- **OAuth2 Integration:** django-allauth for social login (Google, GitHub, etc.)
- **SSO Implementation:** SAML2 for enterprise SSO using djangosaml2
- **Token Storage:** Redis for refresh token whitelist and blacklist

Token Management

```
class JWTAuthentication(BaseAuthentication):
    def authenticate(self, request):
        token = request.COOKIES.get('access_token')
        if not token:
            return None
        try:
            payload = jwt.decode(token, settings.SECRET_KEY)
            user = User.objects.get(id=payload['user_id'])
            return (user, token)
        except (jwt.ExpiredSignatureError, User.DoesNotExist):
            return None
```

Security Considerations

- **Token Rotation:** Issue new refresh token on each use, invalidate old one
- **Blacklisting:** Store revoked tokens in Redis with TTL equal to token expiry
- **CSRF Protection:** Use double-submit cookie pattern for state-changing operations
- **Rate Limiting:** Strict limits on token refresh endpoints to prevent abuse

Stateless Design

- **No Server-Side Sessions:** All state in JWT claims (user_id, roles, permissions)
- **Horizontal Scaling:** Any server can validate tokens without shared state
- **Claims Management:** Include minimal data in JWT, fetch fresh data for sensitive operations
- **Logout:** Client-side token deletion + server-side blacklist for security

10. How would you design a real-time analytics and reporting system in Django that processes millions of events per day with dashboards showing metrics with minimal latency?

Architecture Overview

- **Event Ingestion:** Kafka for high-throughput event streaming, Django writes events asynchronously
- **Stream Processing:** Apache Flink or Kafka Streams for real-time aggregation
- **Storage:** Time-series DB (TimescaleDB/InfluxDB) for metrics, PostgreSQL for dimensional data
- **Caching:** Redis for pre-computed dashboard queries

Data Pipeline

```
class AnalyticsEvent(models.Model):
    event_type = models.CharField(max_length=50, db_index=True)
    user_id = models.IntegerField(db_index=True)
    properties = models.JSONField()
    timestamp = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        indexes = [models.Index(fields=['event_type', 'timestamp'])]
```

Real-Time Processing

- **Lambda Architecture:** Real-time layer (last hour) + Batch layer (historical). Merge results in serving layer.
- **Windowing:** Tumbling windows for counts, sliding windows for moving averages
- **Materialized Views:** Pre-aggregate common queries (daily/hourly rollups) using Celery beat
- **Incremental Updates:** Update aggregates incrementally rather than full recalculation

Dashboard Optimization

- **Query Optimization:** Use OLAP cubes for multi-dimensional analysis
- **Sampling:** For large datasets, use statistical sampling for approximate results
- **WebSocket Updates:** Push real-time metrics to connected dashboards
- **Data Retention:** Hierarchical storage - detailed data for 7 days, aggregated for 90 days, summary forever

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Django view decorator that logs the execution time of any view function.

Solution: Custom Timing Decorator

Here's a reusable decorator that measures and logs view execution time:

```
import time
import logging
from functools import wraps

logger = logging.getLogger(__name__)

def log_execution_time(func):
    @wraps(func)
    def wrapper(request, *args, **kwargs):
        start = time.time()
        response = func(request, *args, **kwargs)
        duration = time.time() - start
        logger.info(f"{func.__name__} took {duration:.2f}s")
        return response
    return wrapper
```

Key points:

- Uses **@wraps** to preserve original function metadata
- Works with any view signature (function-based or class-based with method_decorator)
- Logs using Django's logging framework for production-ready monitoring

2. How would you debug N+1 query problems in Django? Provide a code example showing the problem and solution.

N+1 Query Problem & Solution

Problem: Fetching related objects in a loop causes multiple queries:

```
# Bad: N+1 queries
authors = Author.objects.all()
for author in authors:
    print(author.books.all())  # Query per author!
```

Solution: Use select_related (ForeignKey) or prefetch_related (ManyToMany):

```
# Good: 2 queries total
authors = Author.objects.prefetch_related('books')
for author in authors:
    print(author.books.all())  # No additional queries
```

Debugging tools:

- **django-debug-toolbar**: Visual query inspection in browser
- **connection.queries**: Programmatic query analysis in tests
- **logging**: Enable SQL logging with DEBUG_SQL setting
- **nplusone** package: Automatic N+1 detection

3. Implement a custom Django management command that accepts arguments and handles exceptions properly.

Custom Management Command Structure

```
from django.core.management.base import BaseCommand, CommandError
from myapp.models import User

class Command(BaseCommand):
    help = 'Deactivate users by email domain'

    def add_arguments(self, parser):
        parser.add_argument('domain', type=str)
        parser.add_argument('--dry-run', action='store_true')

    def handle(self, *args, **options):
        domain = options['domain']
        users = User.objects.filter(email__endswith=f'@{domain}')
        if not options['dry_run']:
            users.update(is_active=False)
        self.stdout.write(self.style.SUCCESS(f'Deactivated {users.count()} users'))
```

Best practices:

- Use **CommandError** for expected failures
- Implement **add_arguments** for parameter parsing
- Use **self.stdout.write** instead of print for testability
- Add **--dry-run** flags for safe testing

4. What tools and techniques do you use for memory profiling in Django applications?

Memory Profiling Techniques

1. memory_profiler: Line-by-line memory usage

```
from memory_profiler import profile

@profile
def expensive_view(request):
    large_data = Model.objects.all()
    # Process data
    return response
```

2. Django Debug Toolbar: Shows memory usage per request in development

3. objgraph: Visualize object references and find memory leaks

```
import objgraph
objgraph.show_most_common_types(limit=10)
objgraph.show_growth()  # After operations
```

4. tracemalloc: Built-in Python module for memory tracking

```
import tracemalloc
tracemalloc.start()
# Your code here
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
```

Common issues:

- QuerySet caching holding large datasets
- Circular references preventing garbage collection
- File uploads not using chunked reading

5. Explain monkey patching in Django and provide a practical example where it's useful.

Monkey Patching in Django

Definition: Dynamically modifying classes or modules at runtime to change behavior without altering source code.

Practical example: Extending third-party package behavior:

```
# In apps.py or __init__.py
```

```
from django.contrib.auth.models import User

def get_full_name_with_title(self):
    title = getattr(self, 'title', '')
    return f"{title} {self.first_name} {self.last_name}".strip()

User.add_to_class('get_display_name', get_full_name_with_title)
```

Another use case: Patching for testing:

```
from unittest.mock import patch

@patch('myapp.views.external_api_call')
def test_view(self, mock_api):
    mock_api.return_value = {'status': 'ok'}
    response = self.client.get('/endpoint/')
```

Cautions:

- Use sparingly; prefer inheritance or composition
- Can break with library updates
- Makes code harder to trace and debug
- Best for temporary fixes or testing scenarios

6. How do you handle and debug circular import errors in Django? Provide strategies and code examples.

Circular Import Resolution Strategies

1. Lazy imports inside functions:

```
# models.py
class Author(models.Model):
    def get_top_book(self):
        from .utils import calculate_top_book
        return calculate_top_book(self)
```

2. Use string references in ForeignKey:

```
class Book(models.Model):
    author = models.ForeignKey('myapp.Author', on_delete=models.CASCADE)
    reviewer = models.ForeignKey('auth.User', on_delete=models.SET_NULL)
```

3. Import TYPE_CHECKING for type hints:

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from .models import Author

def process_author(author: 'Author') -> None:
    pass
```

4. Restructure code: Move shared code to separate module

Debugging tips:

- Use **python -v** to see import order
- Check for imports at module level that should be local
- Review signals.py and apps.py for premature imports
- Use **import-linter** to enforce layer architecture

7. Write a context manager for Django database transactions that handles nested transactions and savepoints.

Advanced Transaction Context Manager

```
from django.db import transaction
from contextlib import contextmanager
import logging
```

```
logger = logging.getLogger(__name__)

@contextmanager
def safe_transaction(using=None, savepoint=True):
    try:
        with transaction.atomic(using=using, savepoint=savepoint):
            yield
    except Exception as e:
        logger.error(f"Transaction failed: {e}")
        raise

# Usage:
with safe_transaction():
    User.objects.create(username='test')
    with safe_transaction():  # Nested savepoint
        Profile.objects.create(user=user)
```

Key concepts:

- **atomic()** creates savepoints for nested calls automatically
- **savepoint=False** disables nesting, raising error on nested use
- Use **transaction.on_commit()** for post-transaction actions
- Set **ATOMIC_REQUESTS=True** to wrap all views in transactions

8. How would you implement custom exception handling middleware in Django? Show code and explain the use case.

Custom Exception Handling Middleware

```
import logging
from django.http import JsonResponse
from django.core.exceptions import PermissionDenied

logger = logging.getLogger(__name__)

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

    def __call__(self, request):
        return self.get_response(request)

    def process_exception(self, request, exception):
        logger.error(f"Exception: {exception}", exc_info=True)
        if isinstance(exception, PermissionDenied):
            return JsonResponse({'error': 'Access denied'}, status=403)
        return None
```

Use cases:

- Centralized error logging with context (user, URL, headers)
- Convert exceptions to API-friendly JSON responses
- Integrate with error tracking (Sentry, Rollbar)
- Custom handling for business logic exceptions
- Sanitize error messages before sending to client

Note: Add to MIDDLEWARE setting and ensure it's positioned correctly in the stack.

9. Demonstrate how to use Django's contenttypes framework to create a generic activity log system.

Generic Activity Log with ContentTypes

```
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType
from django.db import models

class ActivityLog(models.Model):
    user = models.ForeignKey('auth.User', on_delete=models.CASCADE)
    action = models.CharField(max_length=50)
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey('content_type', 'object_id')
    timestamp = models.DateTimeField(auto_now_add=True)

# Usage:
ActivityLog.objects.create(user=user, action='created', content_object=book)
```

Benefits:

- Single table tracks activities for any model
- Avoids creating separate log tables per model
- Query logs across different content types

Querying:

```
# Get all logs for a specific object
book_ct = ContentType.objects.get_for_model(Book)
logs = ActivityLog.objects.filter(content_type=book_ct, object_id=book.id)
```

10. What are Django signals and how do you debug issues related to signal handlers not firing or firing multiple times?

Django Signals Debugging

Common issues and solutions:

1. Signal not firing: Handler not registered properly

```
# Correct: In apps.py
class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        import myapp.signals  # Import signals here

# In signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=User)
def user_saved(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

2. Signal firing multiple times: Multiple registrations

- Check if signals.py is imported multiple times
- Use **dispatch_uid** parameter to prevent duplicates

```
post_save.connect(handler, sender=User, dispatch_uid='unique_identifier')
```

3. Debugging signals:

- Add logging inside signal handlers
- Use **django-debug-toolbar** signals panel
- Check **Signal.receivers** list to see registered handlers
- Test signals in isolation with manual .send() calls

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a slow Django application. What was your approach?

Situation: Our e-commerce platform was experiencing 5+ second page load times during peak traffic, causing cart abandonment rates to increase by 30%.

Task: I was tasked with identifying bottlenecks and reducing response times to under 1 second within two weeks.

Action: I used Django Debug Toolbar and django-silk to profile queries, discovering N+1 query problems. I implemented select_related() and prefetch_related() for foreign key relationships, added database indexes on frequently queried fields, implemented Redis caching for product listings, and optimized our ORM queries by using only() and defer() methods.

Result: Page load times dropped to 800ms on average, cart abandonment decreased by 22%, and the application handled 3x more concurrent users without performance degradation.

2. Describe a situation where you had to refactor legacy Django code. How did you ensure the refactoring was successful?

Situation: I inherited a Django 1.8 project with 50,000+ lines of code, no tests, and tightly coupled views containing business logic.

Task: Upgrade to Django 3.2, implement proper architecture, and ensure zero downtime during the transition.

Action: I created a comprehensive test suite achieving 80% coverage before making changes, refactored fat views into service layers and serializers, migrated function-based views to class-based views where appropriate, upgraded dependencies incrementally, and used feature flags to deploy changes gradually. I documented all architectural decisions and conducted code reviews with the team.

Result: Successfully upgraded with zero production incidents, reduced code duplication by 40%, decreased bug reports by 35%, and improved developer onboarding time from 3 weeks to 1 week due to better code organization.

3. Can you share an example of when you had to handle a critical production bug in a Django application?

Situation: On a Friday evening, our payment processing system started failing silently, and customers reported transactions being charged without order confirmations. Approximately 200 transactions were affected.

Task: I needed to identify the root cause immediately, fix the issue, and ensure data integrity for affected transactions.

Action: I reviewed Sentry logs and discovered a race condition in our Celery task handling payment webhooks. I implemented database-level locking using select_for_update(), added idempotency keys to prevent duplicate processing, created a management command to reconcile affected transactions, and set up additional monitoring with custom metrics.

Result: Fixed the issue within 2 hours, successfully reconciled all 200 transactions, implemented automated alerts for similar failures, and prevented potential revenue loss of $50,000+. Documented the incident in a postmortem and shared learnings with the engineering team.

4. Tell me about a time when you had to make a difficult technical decision regarding Django architecture or design patterns.

Situation: Our team was building a multi-tenant SaaS application and debated between using separate databases per tenant versus shared database with tenant isolation via row-level filtering.

Task: As the technical lead, I needed to evaluate both approaches and make a decision that balanced scalability, cost, and development complexity.

Action: I created proof-of-concepts for both approaches, analyzed performance benchmarks with 100+ simulated tenants, calculated infrastructure costs at different scales, evaluated security implications, and assessed developer experience. I used Django's schema-based multi-tenancy with django-tenants middleware for isolation while maintaining a shared database. I presented findings with data to stakeholders.

Result: The chosen approach reduced infrastructure costs by 60% compared to separate databases, simplified backup and maintenance procedures, maintained strong tenant isolation, and allowed us to onboard new tenants in under 30 seconds. The application now serves 500+ tenants successfully.

5. Describe a situation where you had to collaborate with frontend developers or other teams on a Django API project.

Situation: Our mobile app team was frustrated with inconsistent API responses, lack of documentation, and frequent breaking changes from our Django REST Framework API.

Task: I was assigned to improve the API development process and rebuild trust between backend and mobile teams.

Action: I implemented API versioning using DRF's versioning schemes, created comprehensive OpenAPI documentation with drf-spectacular, established a contract-first approach where API specs were reviewed before implementation, set up automated API tests covering all endpoints, and created a staging environment with realistic data. I held weekly sync meetings and created a shared Slack channel for quick communication.

Result: API-related bugs decreased by 70%, mobile team velocity increased by 40%, time spent on integration issues dropped from 15 hours/week to 3 hours/week, and we successfully launched two major features ahead of schedule with zero API-related blockers.

6. Tell me about a time when you implemented a complex feature in Django that required learning new technologies or frameworks.

Situation: Our product team wanted real-time notifications and live updates for a collaborative document editing feature, but our Django application was purely request-response based.

Task: I needed to implement WebSocket support and real-time functionality while maintaining our existing Django infrastructure.

Action: I researched Django Channels and ASGI servers, designed an architecture combining Django Channels with Redis as the channel layer, implemented WebSocket consumers for real-time events, created a hybrid approach where HTTP views handled CRUD operations and WebSockets handled live updates, and thoroughly tested with multiple concurrent users. I documented the new architecture and trained the team.

Result: Successfully delivered real-time collaboration features, handled 1,000+ concurrent WebSocket connections efficiently, reduced perceived latency from 30 seconds (polling) to instant updates, and the feature became our most-praised product enhancement with a 95% satisfaction rating.

7. Share an example of when you had to deal with conflicting requirements or priorities in a Django project.

Situation: Product management wanted to add extensive user activity tracking and analytics, while the security team raised concerns about GDPR compliance and data minimization. Engineering was concerned about database bloat.

Task: I needed to find a solution that satisfied analytics needs while respecting privacy regulations and maintaining system performance.

Action: I organized a meeting with all stakeholders to understand core requirements, designed a privacy-first analytics system using Django signals for event tracking, implemented data anonymization and aggregation at collection time, created automated data retention policies with Django management commands, used time-series database (TimescaleDB) for efficient storage, and ensured explicit user consent mechanisms. I documented privacy impact assessments.

Result: Delivered analytics capabilities that satisfied product requirements, achieved GDPR compliance certified by legal team, reduced storage requirements by 80% through aggregation, and created a reusable framework adopted by other teams. The solution became a template for future privacy-sensitive features.

8. Describe a time when you mentored junior developers or led a code review that significantly improved code quality.

Situation: Our team hired three junior Django developers who were producing code with security vulnerabilities, poor ORM usage, and inconsistent patterns, leading to increased bug rates and slower PR reviews.

Task: I was asked to mentor the junior developers and establish better coding standards across the team.

Action: I created a Django best practices guide specific to our codebase, conducted weekly code review sessions explaining not just what was wrong but why, implemented automated linting with pylint and flake8 with custom rules, created code review checklists focusing on security, performance, and Django idioms, pair-programmed on complex features, and encouraged questions in a judgment-free environment. I also set up pre-commit hooks to catch common issues.

Result: Within three months, junior developers were writing production-ready code with minimal revisions, code review time decreased by 50%, security vulnerabilities in new code dropped to near zero, and two junior developers successfully led feature implementations independently. One mentee later became a team lead.

9. Tell me about a time when you had to balance technical debt with feature development in a Django project.

Situation: Our Django application had accumulated significant technical debt including Django 2.2 (near end-of-life), 30% test coverage, and a monolithic architecture, while business pressure demanded new features for competitive reasons.

Task: I needed to create a strategy that addressed technical debt without halting feature development or risking system stability.

Action: I proposed and implemented a 70-30 rule: 70% sprint capacity for features, 30% for technical debt. I prioritized debt by risk and impact, tackled the Django upgrade incrementally across sprints, implemented the Boy Scout Rule (leave code better than you found it), created automated dependency updates with Dependabot, and made technical debt visible in sprint planning. I also refactored code opportunistically when working on related features.

Result: Successfully upgraded to Django 4.1 over four months without disrupting feature delivery, increased test coverage to 75%, reduced critical security vulnerabilities by 100%, improved deployment time from 45 minutes to 10 minutes, and maintained feature delivery velocity. Management recognized this as a model for sustainable development.

10. Describe a situation where you had to debug a complex issue in Django that involved multiple layers of the application stack.

Situation: Users reported intermittent 500 errors occurring randomly across different endpoints, with no clear pattern. Error rates spiked to 5% during business hours, affecting customer trust.

Task: I needed to identify the root cause of these sporadic failures that weren't consistently reproducible in our staging environment.

Action: I enhanced logging with structured logging using django-structlog, added distributed tracing with OpenTelemetry to track requests across services, analyzed database connection pool metrics and discovered pool exhaustion, investigated Celery task queues and found memory leaks in long-running tasks, reviewed middleware chain for potential issues, and used Django's connection management to implement proper connection handling. I created a load testing scenario that reproduced the issue.

Result: Identified that a third-party library was keeping database connections open indefinitely. Fixed by implementing connection pooling with proper timeouts and upgrading the problematic library. Error rates dropped to 0.1%, established better observability practices, created runbooks for similar issues, and prevented potential customer churn estimated at $200,000 annually.