Django

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.

Django Request-Response Cycle

The Django request-response cycle follows a specific execution path:

- **Request Phase:** HttpRequest object is created and passes through middleware's process_request() methods (top to bottom)
- **URL Resolution:** URLconf determines the view function
- **View Middleware:** process_view() middleware executes before the view
- **View Execution:** View function processes the request and returns HttpResponse
- **Template Rendering:** If using templates, process_template_response() middleware executes
- **Response Phase:** process_response() middleware executes (bottom to top)
- **Exception Handling:** process_exception() runs if any exception occurs

Middleware order matters: Request processing goes top-down, response processing goes bottom-up. SecurityMiddleware should be early, while CommonMiddleware can be later.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    # Custom middleware placement is critical
]
```

2. How does Django's ORM handle N+1 query problems, and what are the best practices to avoid them?

N+1 Query Problem and Solutions

The **N+1 problem** occurs when code executes one query to fetch N objects, then N additional queries to fetch related objects.

Detection:

- Use Django Debug Toolbar or django.db.connection.queries
- Enable SQL logging in development
- Monitor database query counts in tests

Solutions:

- **select_related():** For ForeignKey and OneToOne (SQL JOIN)
- **prefetch_related():** For ManyToMany and reverse ForeignKey (separate queries with Python joining)
- **Prefetch objects:** Custom prefetch queries with filtering

```
# Bad: N+1 queries
for book in Book.objects.all():
    print(book.author.name)

# Good: 1 query with JOIN
books = Book.objects.select_related('author')

# Good: 2 queries total
books = Book.objects.prefetch_related('tags')
```

Advanced: Use Prefetch() objects for conditional prefetching and only()/defer() to limit field selection.

3. What are Django signals, and when should you avoid using them?

Django Signals

Signals allow decoupled applications to get notified when actions occur elsewhere in the framework. Built-in signals include pre_save, post_save, pre_delete, post_delete, and m2m_changed.

Common Use Cases:

- Creating related objects (user profiles after user creation)
- Clearing caches when models change
- Logging or auditing changes
- Sending notifications

When to Avoid Signals:

- **Implicit behavior:** Makes code harder to trace and debug
- **Performance:** Adds overhead and can cause unexpected database queries
- **Transaction issues:** Signals fire even if transaction rolls back (use transaction.on_commit())
- **Testing complexity:** Hard to isolate and mock
- **Bulk operations:** Signals don't fire for bulk_create(), update(), or delete() querysets

Better alternatives: Override model save() method, use custom managers, or explicit service layer methods.

```
# Prefer explicit over implicit
class User(models.Model):
    def save(self, *args, **kwargs):
        super().save(*args, **kwargs)
        self.create_profile()
```

4. Explain Django's database transaction management and isolation levels.

Transaction Management in Django

Django provides multiple ways to manage database transactions:

1. ATOMIC_REQUESTS:

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

Wraps each view in a transaction. Commits on success, rolls back on exception.

2. atomic() Decorator/Context Manager:

```
from django.db import transaction

@transaction.atomic
def my_view(request):
    # All DB operations in one transaction
    pass
```

3. Savepoints for Nested Transactions:

```
with transaction.atomic():
    create_parent()
    try:
        with transaction.atomic():
            create_child()  # May fail
    except IntegrityError:
        pass  # Parent still commits
```

Isolation Levels: Django uses database defaults (usually READ COMMITTED). For custom isolation:

```
from django.db import transaction
transaction.set_isolation_level(
    transaction.ISOLATION_LEVEL_SERIALIZABLE
)
```

Best practices: Use select_for_update() for row-level locking, avoid long transactions, and use transaction.on_commit() for post-commit hooks.

5. How do Django's class-based views (CBVs) work internally, and what is the method resolution order?

Class-Based Views Architecture

CBVs use object-oriented patterns with mixins and inheritance. The core is the View class with an as_view() class method.

Execution Flow:

- as_view() returns a view function
- View function calls dispatch() method
- dispatch() routes to HTTP method handlers (get(), post(), etc.)
- Method handler processes request and returns response

Method Resolution Order (MRO):

Python's C3 linearization determines method lookup. Django uses left-to-right, depth-first order:

```
class MyView(LoginRequiredMixin,
             PermissionRequiredMixin,
             DetailView):
    # MRO: MyView → LoginRequiredMixin →
    # PermissionRequiredMixin → DetailView →
    # BaseDetailView → SingleObjectMixin → View
```

Key Methods in Generic Views:

- setup(): Initialize view attributes
- dispatch(): Route to HTTP method
- get_queryset(): Return base queryset
- get_object(): Retrieve single object
- get_context_data(): Build template context

Mixin order matters: Place authentication/permission mixins before view classes.

6. What are Django's caching strategies, and how do you implement cache invalidation?

Django Caching Strategies

Django supports multiple caching levels:

1. Per-Site Cache: Caches entire site output

```
MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    # Other middleware
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

2. Per-View Cache:

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

@cache_page(60 * 15)  # 15 minutes
def my_view(request):
    return render(request, 'template.html')
```

3. Template Fragment Cache:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
```

```
    .. expensive sidebar ..
{% endcache %}
```

4. Low-Level Cache API:

```
from django.core.cache import cache

data = cache.get('my_key')
if data is None:
    data = expensive_query()
    cache.set('my_key', data, 300)
```

Cache Invalidation Strategies:

- **Time-based:** Set appropriate TTL values
- **Event-based:** Invalidate on model save using signals or overridden save()
- **Version-based:** Use cache key versioning
- **Tag-based:** Use Redis or Memcached with tag support

Backends: Redis (recommended), Memcached, Database, File-based, or Local-memory.

7. Explain Django's middleware architecture and how to create custom middleware.

Django Middleware Architecture

Middleware is a framework of hooks into Django's request/response processing. Each middleware component is responsible for a specific function.

Modern Middleware (Django 1.10+):

```
class CustomMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration

    def __call__(self, request):
        # Code before view
        response = self.get_response(request)
        # Code after view
        return response
```

Additional Hooks:

- process_view(request, view_func, view_args, view_kwargs)
- process_exception(request, exception)
- process_template_response(request, response)

Example: Request Timing Middleware:

```
import time

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

    def __call__(self, request):
        start = time.time()
        response = self.get_response(request)
        duration = time.time() - start
        response['X-Response-Time'] = duration
        return response
```

Best Practices: Keep middleware lightweight, avoid database queries, use async middleware for async views, and order middleware correctly.

8. How does Django handle database migrations, and what are the best practices for managing them in production?

Django Migrations System

Django's migration system is a version control for database schema. It tracks changes and applies them systematically.

Migration Components:

- **Migration files:** Python files defining operations
- **django_migrations table:** Tracks applied migrations
- **Operations:** CreateModel, AddField, RunSQL, RunPython, etc.

Key Commands:

```
# Create migrations
python manage.py makemigrations

# Apply migrations
python manage.py migrate

# Show migration status
python manage.py showmigrations

# Generate SQL without applying
python manage.py sqlmigrate app_name 0001
```

Production Best Practices:

- **Zero-downtime deployments:** Make backward-compatible changes
- **Split risky migrations:** Separate schema and data migrations
- **Test migrations:** Test both forward and backward (rollback)
- **Avoid RunPython on large tables:** Use data migration scripts instead
- **Use --fake for manual changes:** Sync migration state without applying
- **Squash migrations periodically:** Reduce migration count

Safe migration pattern:

1. Add new field (nullable)
2. Deploy code using new field
3. Backfill data
4. Make field non-nullable
5. Remove old field

9. What are Django's security features, and how do you implement protection against common vulnerabilities?

Django Security Features

Django provides built-in protection against common web vulnerabilities:

1. CSRF Protection:

- Automatic CSRF token validation for POST requests
- Use {% csrf_token %} in forms
- For AJAX: Include X-CSRFToken header

2. SQL Injection Protection:

- ORM automatically escapes parameters
- Use parameterized queries: Model.objects.raw('SELECT * FROM table WHERE id = %s', [id])
- Never use string formatting for queries

3. XSS Protection:

- Template engine auto-escapes variables
- Use |safe filter cautiously
- Sanitize user input with libraries like bleach

4. Clickjacking Protection:

```
MIDDLEWARE = [
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
X_FRAME_OPTIONS = 'DENY'
```

5. HTTPS/SSL:

```
SECURE_SSL_REDIRECT = True
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

6. Additional Settings:

```
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_BROWSER_XSS_FILTER = True
SECURE_REFERRER_POLICY = 'same-origin'
```

Best practices: Run python manage.py check --deploy, keep Django updated, use security-focused middleware, and implement rate limiting.

10. Explain Django's async views and ORM support. What are the limitations and best practices?

Django Async Support

Django 3.0+ supports ASGI and async views. Django 4.1+ added async ORM support.

Async Views:

```
async def my_async_view(request):
    data = await fetch_data()
    return JsonResponse(data)

# Class-based
from django.views import View

class MyAsyncView(View):
    async def get(self, request):
        data = await async_operation()
        return JsonResponse(data)
```

Async ORM (Django 4.1+):

```
async def get_users():
    users = await User.objects.all().acount()
    async for user in User.objects.all():
        print(user.username)

    user = await User.objects.aget(pk=1)
    await user.asave()
```

Limitations:

- Not all ORM operations have async equivalents yet
- Middleware must be async-compatible
- Template rendering is still synchronous
- Some third-party packages lack async support
- Database drivers must support async (asyncpg for PostgreSQL)

Best Practices:

- Use async views for I/O-bound operations (API calls, file operations)
- Don't mix sync and async ORM in same function
- Use sync_to_async() and async_to_sync() for compatibility
- Configure ASGI server (Uvicorn, Daphne)
- Monitor for blocking operations

Data Structures and Algorithms

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

1. How would you implement an LRU (Least Recently Used) cache in Python for a Django application?

LRU Cache Implementation: An LRU cache can be efficiently implemented using a combination of a dictionary and a doubly linked list, or using Python's OrderedDict. Here's a clean implementation:

```
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). In Django, this is useful for caching query results or session data.

2. Explain how Django's QuerySet uses lazy evaluation and what data structure optimizations occur under the hood.

QuerySet Lazy Evaluation: Django QuerySets use lazy evaluation, meaning the database query isn't executed until the data is actually needed. Internally, QuerySets maintain:

- **Query object:** Stores SQL construction details using a tree-like structure
- **Result cache:** A list that stores fetched results to avoid redundant queries
- **Iterator protocol:** Allows efficient iteration without loading all data into memory

```
# Query is not executed yet
users = User.objects.filter(is_active=True)

# Query executes here (first access)
for user in users:
    print(user.name)

# Uses cached results (no new query)
count = len(users)
```

Optimization: The result cache uses a Python list internally, providing O(1) access after initial fetch. Use **iterator()** method for large datasets to avoid caching.

3. Implement a function to find all pairs in a list that sum to a target value, optimized for Django model querysets.

Pair Sum Problem: For a list of integers, find all unique pairs that sum to a target. The optimal approach uses a hash set for O(n) time complexity:

```
def find_pairs_sum(numbers, target):
    seen = set()
    pairs = set()

    for num in numbers:
        complement = target - num
        if complement in seen:
            pairs.add((min(num, complement), max(num, complement)))
        seen.add(num)

    return list(pairs)
```

Django QuerySet Application: For database queries, use aggregation:

```
from django.db.models import F

# Find product pairs where price1 + price2 = target
Product.objects.filter(
    price__lte=target
).annotate(
    complement=target - F('price')
)
```

Complexity: O(n) time, O(n) space.

4. How would you implement a sliding window algorithm for rate limiting in Django?

Sliding Window Rate Limiter: A sliding window tracks requests within a time frame using a deque or sorted list. Here's an implementation:

```
from collections import deque
from time import time

class SlidingWindowRateLimiter:
    def __init__(self, max_requests, window_seconds):
        self.max_requests = max_requests
        self.window = window_seconds
        self.requests = deque()

    def allow_request(self):
        now = time()
        while self.requests and self.requests[0] <= now - self.window:
            self.requests.popleft()

        if len(self.requests) < self.max_requests:
            self.requests.append(now)
            return True
        return False
```

Django Integration: Use with middleware or decorators. Store in Redis for distributed systems.
Time Complexity: O(1) amortized for cleanup, O(1) for check.

5. Explain the time complexity of Django's select_related() vs prefetch_related() and when to use each.

select_related(): Uses SQL JOIN to fetch related objects in a single query. Works with ForeignKey and OneToOne relationships.

```
# Single query with JOIN
books = Book.objects.select_related('author').all()
for book in books:
    print(book.author.name)  # No additional query
```

Time Complexity: O(n) for n records, single database hit. **prefetch_related():** Performs separate queries and joins in Python. Works with ManyToMany and reverse ForeignKey.

```
# Two queries: one for authors, one for books
authors = Author.objects.prefetch_related('books').all()
for author in authors:
    for book in author.books.all():  # Uses cached data
        print(book.title)
```

Time Complexity: O(n + m) where m is related objects. **Use select_related** for single-valued relationships, **prefetch_related** for multi-valued.

6. Implement a Trie data structure for autocomplete functionality in a Django search feature.

Trie Implementation: A Trie (prefix tree) efficiently stores and searches strings with common prefixes:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search_prefix(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]
        return self._collect_words(node, prefix)
```

Complexity: Insert O(m), Search O(m + k) where m=word length, k=results. **Django Use:** Cache Trie in Redis for fast autocomplete.

7. How do you optimize a Django query that performs multiple lookups? Discuss the use of dictionaries and hash tables.

Query Optimization with Hash Tables: When performing multiple lookups, convert QuerySets to dictionaries for O(1) access instead of O(n) repeated queries:

```
# Inefficient: O(n*m) - n users, m lookups
for order in orders:
    user = User.objects.get(id=order.user_id)  # Query each time

# Optimized: O(n+m) using dictionary
user_ids = [order.user_id for order in orders]
users_dict = {u.id: u for u in User.objects.filter(id__in=user_ids)}

for order in orders:
    user = users_dict[order.user_id]  # O(1) lookup
```

Hash Table Benefits:

- Reduces database queries from O(n) to O(1)
- Python dict uses hash table: average O(1) lookup

- Memory trade-off: O(n) space for O(n) time improvement

8. Implement a binary search algorithm for paginated Django QuerySets to find a specific record efficiently.

Binary Search on Ordered QuerySet: When searching ordered data, binary search provides O(log n) complexity:

```
def binary_search_queryset(model, field, target):
    queryset = model.objects.order_by(field)
    left, right = 0, queryset.count() - 1

    while left <= right:
        mid = (left + right) // 2
        mid_value = getattr(queryset[mid], field)

        if mid_value == target:
            return queryset[mid]
        elif mid_value < target:
            left = mid + 1
        else:
            right = mid - 1
    return None
```

Caveat: QuerySet slicing triggers database queries. For large datasets, use database-level binary search or indexing. **Better approach:** Use indexed fields with filter() which leverages B-tree indexes (O(log n)).

9. Explain how to implement a priority queue for task scheduling in Django using heapq.

Priority Queue with heapq: Python's heapq provides a min-heap implementation for priority queue operations:

```
import heapq
from datetime import datetime

class TaskScheduler:
    def __init__(self):
        self.heap = []

    def add_task(self, priority, task_id, task_data):
        heapq.heappush(self.heap, (priority, task_id, task_data))

    def get_next_task(self):
        if self.heap:
            return heapq.heappop(self.heap)
        return None

    def peek(self):
        return self.heap[0] if self.heap else None
```

Time Complexity: Push O(log n), Pop O(log n), Peek O(1). **Django Integration:** Use with Celery for task prioritization or implement custom task queue. Store in Redis for distributed systems.

10. Design a solution to detect N+1 query problems in Django using algorithmic analysis and data structures.

N+1 Query Detection: N+1 problems occur when iterating over objects triggers additional queries. Detect using query counting and set-based analysis:

```
from django.db import connection, reset_queries
from django.conf import settings

class QueryAnalyzer:
    def __init__(self):
        settings.DEBUG = True
```

```
        self.query_patterns = set()

    def analyze(self, queryset):
        reset_queries()
        list(queryset)  # Force evaluation
        queries = connection.queries

        # Detect repeated similar queries
        for q in queries:
            pattern = self._normalize_query(q['sql'])
            if pattern in self.query_patterns:
                return f"N+1 detected: {pattern}"
            self.query_patterns.add(pattern)
```

Solution: Use select_related/prefetch_related. **Tool:** django-debug-toolbar shows query counts. **Algorithm:** Set-based duplicate detection runs in O(n) time.

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 decisions you would make?

Key Architectural Decisions

1. URL Generation Strategy:

- Use base62 encoding (a-z, A-Z, 0-9) for short codes
- Generate 6-7 character codes providing 62^7 = 3.5 trillion URLs
- Use counter-based or hash-based approach with collision handling

2. Database Design:

- Primary table: id (bigint), short_code (varchar indexed), original_url, created_at, expires_at
- Use PostgreSQL with proper indexing on short_code
- Consider sharding by hash of short_code for horizontal scaling

3. Caching Layer:

- Redis for hot URLs (80/20 rule applies)
- Cache short_code -> original_url mappings
- TTL-based expiration aligned with URL expiry

4. Django Implementation:

```
class URLMapping(models.Model):
    short_code = models.CharField(max_length=7, unique=True, db_index=True)
    original_url = models.URLField(max_length=2048)
    created_at = models.DateTimeField(auto_now_add=True)
    expires_at = models.DateTimeField(null=True)
    clicks = models.BigIntegerField(default=0)
```

5. Scalability Considerations:

- Stateless Django application servers behind load balancer
- Read replicas for analytics queries
- CDN for redirect responses with proper cache headers
- Async task queue (Celery) for analytics processing

6. High Availability:

- Multi-AZ database deployment
- Redis Sentinel or Redis Cluster for cache HA
- Rate limiting per IP using Django middleware + Redis

2. How would you design a real-time notification system in Django supporting millions of users?

Real-Time Notification Architecture

1. Technology Stack:

- Django Channels with ASGI server (Daphne/Uvicorn)
- Redis as channel layer and message broker
- WebSocket for real-time push
- Fallback to Server-Sent Events (SSE) or long polling

2. Database Schema:

```
class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=CASCADE)
    type = models.CharField(max_length=50)
    content = models.JSONField()
    is_read = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        indexes = [models.Index(fields=['user', '-created_at'])]
```

3. WebSocket Consumer:

```
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()
```

4. Scalability Strategy:

- Horizontal scaling of ASGI servers
- Redis Cluster for channel layer distribution
- Connection management: track active connections per server
- Use presence detection with heartbeat mechanism

5. Notification Delivery:

- Immediate: WebSocket push for online users
- Deferred: Store in DB, deliver on reconnection
- Batch processing: Celery tasks for bulk notifications
- Priority queue: Critical notifications get precedence

6. Performance Optimizations:

- Pagination for notification history
- Aggregate unread counts in Redis
- Lazy loading with infinite scroll
- Compress WebSocket messages

7. Reliability:

- Message acknowledgment protocol
- Retry mechanism for failed deliveries
- Dead letter queue for persistent failures

3. Design a social media feed system in Django. How would you handle feed generation, ranking, and real-time updates?

Social Media Feed Architecture

1. Feed Generation Approaches:

- **Fan-out on Write (Push):** Pre-compute feeds when posts are created
- **Fan-out on Read (Pull):** Generate feed on user request
- **Hybrid:** Push for active users, pull for inactive/celebrity accounts

2. Database Schema:

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

class Feed(models.Model):
    user = models.ForeignKey(User, on_delete=CASCADE)
    post = models.ForeignKey(Post, on_delete=CASCADE)
    score = models.FloatField()
    created_at = models.DateTimeField()
```

3. Feed Ranking Algorithm:

- Time decay: newer posts rank higher
- Engagement signals: likes, comments, shares
- User affinity: interaction history with author
- Content type preferences
- ML-based personalization with feature vectors

4. Implementation Strategy:

- Use Celery for async feed fan-out on post creation
- Redis Sorted Sets for timeline storage (score = timestamp + ranking)
- Cache top N posts per user (N=100-500)
- PostgreSQL for persistent storage and backfill

5. Real-Time Updates:

- WebSocket notifications for new posts
- Incremental feed updates using cursor-based pagination
- Optimistic UI updates with eventual consistency

6. Scalability Patterns:

- Partition feeds by user_id hash
- Read replicas for feed queries
- CDN caching for media content
- Rate limiting on feed refresh

7. Performance Optimization:

- Denormalize author info in feed entries
- Batch database queries using select_related/prefetch_related
- Lazy load media URLs
- Implement infinite scroll with cursor pagination

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

Multi-Tenancy Design Patterns

1. Isolation Strategies:

- **Shared Database, Shared Schema:** tenant_id column in all tables
- **Shared Database, Separate Schema:** PostgreSQL schemas per tenant
- **Separate Database:** Complete isolation, higher overhead

2. Recommended Approach (Shared DB, Shared Schema):

```
class TenantModel(models.Model):
    tenant = models.ForeignKey('Tenant', on_delete=CASCADE)
    class Meta:
        abstract = True
        indexes = [models.Index(fields=['tenant'])]

class Customer(TenantModel):
    name = models.CharField(max_length=255)
    email = models.EmailField()
```

3. Middleware for Tenant Context:

```
class TenantMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
    def __call__(self, request):
        tenant = self.get_tenant(request)
        request.tenant = tenant
        return self.get_response(request)
```

4. Query Filtering:

- Custom manager to auto-filter by tenant
- Use django-tenant-schemas or django-tenants library

- Override get_queryset() in managers
- Database-level Row Level Security (RLS) in PostgreSQL

5. Performance Considerations:

- Composite indexes: (tenant_id, created_at)
- Partition tables by tenant_id for large datasets
- Separate Redis namespaces per tenant
- Connection pooling with pgbouncer

6. Security Measures:

- Validate tenant ownership in all views/APIs
- Prevent cross-tenant data leakage in querysets
- Tenant-specific encryption keys
- Audit logging per tenant

7. Scalability:

- Shard large tenants to dedicated databases
- Background jobs with tenant context
- Tenant-aware caching strategies
- Resource quotas and rate limiting per tenant

5. Design a distributed task processing system using Django and Celery. How would you ensure reliability, ordering, and scalability?

Distributed Task Processing Architecture

1. Celery Configuration:

```
CELERY_BROKER_URL = 'redis://localhost:6379/0'
CELERY_RESULT_BACKEND = 'redis://localhost:6379/1'
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TASK_TRACK_STARTED = True
CELERY_TASK_TIME_LIMIT = 30 * 60
CELERY_TASK_ACKS_LATE = True
```

2. Task Design Patterns:

- **Idempotency:** Tasks should be safely retryable
- **Atomicity:** Use database transactions
- **Immutability:** Pass IDs, not objects
- Use task signatures for complex workflows

3. Reliability Mechanisms:

- Task acknowledgment: CELERY_TASK_ACKS_LATE = True
- Retry with exponential backoff
- Dead letter queues for failed tasks
- Task state persistence in result backend

4. Ordering Guarantees:

- Use priority queues with task routing
- Implement task chains for sequential execution
- Single worker per queue for strict ordering
- Database-based locking for critical sections

5. Scalability Strategy:

- Multiple queues by priority/type
- Auto-scaling workers based on queue depth
- Rate limiting: @task(rate_limit='100/m')
- Task routing to specialized workers

6. Monitoring and Observability:

- Flower for real-time monitoring
- Prometheus metrics export

- Task execution time tracking
- Alert on queue buildup or failure rate

7. Advanced Patterns:

- Chord: parallel tasks with callback
- Chain: sequential task execution
- Group: parallel task execution
- Canvas for complex workflows

8. Failure Handling:

```
@shared_task(bind=True, max_retries=3)
def process_order(self, order_id):
    try:
        order = Order.objects.get(id=order_id)
        # processing logic
    except Exception as exc:
        raise self.retry(exc=exc, countdown=60)
```

6. How would you implement a high-performance search system in Django supporting full-text search, filtering, and faceting?

Search System Architecture

1. Technology Choices:

- **Elasticsearch:** Best for full-text, complex queries, analytics
- **PostgreSQL Full-Text Search:** Good for simple cases, reduces infrastructure
- **Solr:** Alternative to Elasticsearch
- **Typesense/Meilisearch:** Lightweight alternatives

2. Elasticsearch Integration:

```
from elasticsearch_dsl import Document, Text, Keyword

class ProductDocument(Document):
    title = Text(analyzer='snowball')
    description = Text()
    category = Keyword()
    price = Float()
    class Index:
        name = 'products'
```

3. Indexing Strategy:

- Sync data on save using Django signals
- Bulk indexing with management command
- Use Celery for async indexing to avoid blocking
- Implement delta indexing for updates

4. Search Implementation:

- Query DSL for complex searches
- Aggregations for faceted navigation
- Highlighting for search result snippets
- Fuzzy matching for typo tolerance
- Boosting for relevance tuning

5. Performance Optimization:

- Cache frequent queries in Redis (TTL 5-15 min)
- Use search-as-you-type with debouncing
- Implement pagination with search_after
- Filter before query for better performance
- Use routing for multi-tenant scenarios

6. PostgreSQL Full-Text Alternative:

```
class Product(models.Model):
```

```
    search_vector = SearchVectorField(null=True)
    class Meta:
        indexes = [GinIndex(fields=['search_vector'])]

# Query
Product.objects.annotate(
    rank=SearchRank(F('search_vector'), query)
).filter(search_vector=query).order_by('-rank')
```

7. Faceted Search:

- Aggregations for category counts
- Price range buckets
- Dynamic facets based on query context
- Multi-select facets with filter combination

8. Scalability:

- Elasticsearch cluster with replica shards
- Load balancing across cluster nodes
- Index aliasing for zero-downtime reindexing
- Hot-warm architecture for time-series data

7. Design an API rate limiting and throttling system for a Django REST API. How would you handle different tiers and distributed environments?

Rate Limiting Architecture

1. Rate Limiting Strategies:

- **Fixed Window:** Simple counter per time window
- **Sliding Window:** More accurate, higher complexity
- **Token Bucket:** Allows bursts, smooth rate limiting
- **Leaky Bucket:** Constant output rate

2. Django REST Framework Implementation:

```
from rest_framework.throttling import UserRateThrottle

class BurstRateThrottle(UserRateThrottle):
    scope = 'burst'
    rate = '60/min'

class SustainedRateThrottle(UserRateThrottle):
    scope = 'sustained'
    rate = '1000/day'
```

3. Redis-Based Implementation:

- Use Redis for distributed rate limiting
- Atomic operations with Lua scripts
- Key pattern: rate_limit:{user_id}:{endpoint}:{window}
- TTL for automatic cleanup

4. Token Bucket Algorithm:

```
def check_rate_limit(user_id, capacity, refill_rate):
    key = f'bucket:{user_id}'
    now = time.time()
    tokens, last_refill = redis.hmget(key, 'tokens', 'last')
    elapsed = now - float(last_refill or now)
    tokens = min(capacity, float(tokens or capacity) + elapsed * refill_rate)
    if tokens >= 1:
        redis.hset(key, 'tokens', tokens - 1)
        return True
    return False
```

5. Multi-Tier System:

- Free tier: 100 requests/hour

- Pro tier: 1000 requests/hour
- Enterprise: Custom limits or unlimited
- Store tier info in user profile or JWT claims

6. Distributed Considerations:

- Centralized Redis for consistent limits
- Redis Cluster for high availability
- Handle Redis failures gracefully (fail open vs closed)
- Local cache with eventual consistency for read-heavy

7. Response Headers:

- X-RateLimit-Limit: Total allowed requests
- X-RateLimit-Remaining: Requests remaining
- X-RateLimit-Reset: Timestamp for limit reset
- Retry-After: Seconds to wait when limited

8. Advanced Features:

- Per-endpoint rate limits
- IP-based limiting for anonymous users
- Whitelist for internal services
- Dynamic limits based on system load
- Cost-based limiting (expensive operations count more)

8. How would you design a file upload and processing system in Django handling large files, virus scanning, and thumbnail generation?

File Upload System Architecture

1. Upload Strategy:

- **Direct Upload:** Client -> Django -> Storage (simple, blocks worker)
- **Pre-signed URLs:** Client -> S3 directly (recommended for large files)
- **Chunked Upload:** Resume capability for large files
- **Multipart Upload:** S3 multipart for files >100MB

2. Storage Backend:

```
AWS_STORAGE_BUCKET_NAME = 'my-bucket'
AWS_S3_CUSTOM_DOMAIN = f'{AWS_STORAGE_BUCKET_NAME}.s3.amazonaws.com'
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
AWS_S3_OBJECT_PARAMETERS = {
    'CacheControl': 'max-age=86400',
}
AWS_DEFAULT_ACL = 'private'
```

3. Upload Model:

```
class Upload(models.Model):
    user = models.ForeignKey(User, on_delete=CASCADE)
    file = models.FileField(upload_to='uploads/%Y/%m/')
    status = models.CharField(max_length=20)
    file_size = models.BigIntegerField()
    content_type = models.CharField(max_length=100)
    virus_scan_status = models.CharField(max_length=20)
```

4. Pre-signed URL Generation:

- Generate temporary upload URL (15-60 min expiry)
- Client uploads directly to S3
- S3 event triggers Lambda/webhook to Django
- Django creates database record and queues processing

5. Virus Scanning:

- ClamAV integration via pyclamd
- Scan in Celery task before making file public
- Quarantine suspicious files

- S3 bucket scanning with Lambda + ClamAV

6. Image Processing Pipeline:

- Celery task for async processing
- Generate multiple thumbnail sizes
- Use Pillow or ImageMagick
- Store in separate S3 prefix
- Update database with thumbnail URLs

7. Processing Task:

```
@shared_task
def process_upload(upload_id):
    upload = Upload.objects.get(id=upload_id)
    scan_virus(upload.file.path)
    if upload.content_type.startswith('image'):
        generate_thumbnails(upload)
    upload.status = 'completed'
    upload.save()
```

8. Scalability & Performance:

- CDN (CloudFront) for file delivery
- Separate buckets for uploads and processed files
- Lifecycle policies for cleanup
- Progress tracking via WebSocket
- Rate limiting on uploads

9. Security:

- Validate file extensions and MIME types
- Size limits per user tier
- Generate random filenames to prevent enumeration
- Signed URLs for private file access

9. Design a session management and authentication system for a Django application supporting multiple authentication methods and distributed sessions.

Authentication & Session Architecture

1. Authentication Methods:

- Username/Password with Django auth
- OAuth2 (Google, GitHub, etc.) via django-allauth
- JWT tokens for API authentication
- SAML for enterprise SSO
- Two-factor authentication (TOTP)

2. JWT Implementation:

```
from rest_framework_simplejwt.tokens import RefreshToken

def get_tokens_for_user(user):
    refresh = RefreshToken.for_user(user)
    return {
        'refresh': str(refresh),
        'access': str(refresh.access_token),
    }
```

3. Session Backend Options:

- **Database:** Default, not scalable
- **Cache (Redis):** Fast, recommended for production
- **Signed Cookies:** Stateless, limited size
- **Hybrid:** Redis with DB fallback

4. Redis Session Configuration:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
```

```
SESSION_CACHE_ALIAS = 'default'
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
    }
}
```

5. Multi-Factor Authentication:

- Use django-otp or django-two-factor-auth
- TOTP (Time-based One-Time Password)
- SMS/Email verification codes
- Backup codes for recovery
- Remember trusted devices

6. Distributed Session Management:

- Redis Cluster for high availability
- Session replication across regions
- Sticky sessions at load balancer (optional)
- Session migration on Redis failover

7. Security Best Practices:

- HTTPS only for cookies (SECURE=True)
- HttpOnly cookies to prevent XSS
- SameSite=Strict for CSRF protection
- Short session timeout for sensitive operations
- Rotate session ID on privilege escalation

8. Token Management:

- Short-lived access tokens (15 min)
- Long-lived refresh tokens (7-30 days)
- Token rotation on refresh
- Blacklist for revoked tokens in Redis
- Store refresh tokens hashed in database

9. SSO Integration:

- django-allauth for social authentication
- python3-saml for SAML SSO
- Centralized user provisioning
- Just-in-time user creation

10. Monitoring:

- Track failed login attempts
- Account lockout after N failures
- Audit log for authentication events
- Alert on suspicious patterns

10. How would you design a real-time analytics and reporting system in Django for processing millions of events per day?

Real-Time Analytics Architecture

1. Data Ingestion Layer:

- **API Endpoint:** Django view for event submission
- **Message Queue:** Kafka or RabbitMQ for buffering
- **Batch API:** Accept multiple events in single request
- **Edge Processing:** Validate and enrich at ingestion

2. Event Model:

```
class Event(models.Model):
    event_type = models.CharField(max_length=50, db_index=True)
    user_id = models.BigIntegerField(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'])]
```

3. Processing Pipeline:

- Celery workers consume from queue
- Validate and transform events
- Write to time-series database (InfluxDB/TimescaleDB)
- Update aggregates in Redis
- Archive to data warehouse (S3 + Athena/BigQuery)

4. Time-Series Storage:

- **TimescaleDB:** PostgreSQL extension for time-series
- **InfluxDB:** Purpose-built time-series DB
- **Clickhouse:** Column-oriented for analytics
- Partition by time (daily/weekly)
- Retention policies for old data

5. Real-Time Aggregation:

```
# Redis for real-time counters
def track_event(event_type, user_id):
    key = f'events:{event_type}:{date.today()}'
    redis.incr(key)
    redis.expire(key, 86400 * 7)
    redis.pfadd(f'unique_users:{date.today()}', user_id)
```

6. Reporting Layer:

- Pre-compute common aggregates (hourly/daily)
- Materialized views for complex queries
- Cache dashboard data (5-15 min TTL)
- Use Django ORM with aggregations for simple reports
- Direct SQL for complex analytics

7. Query Optimization:

- Time-based partitioning
- Columnar storage for analytical queries
- Index on common filter columns
- Use OLAP cubes for multi-dimensional analysis
- Query result caching

8. Scalability Patterns:

- Horizontal scaling of ingestion API
- Kafka partitioning by user_id or event_type
- Separate read/write databases
- CDC (Change Data Capture) for real-time sync
- Lambda architecture: batch + stream processing

9. Data Warehouse Integration:

- Periodic export to S3 (Parquet format)
- AWS Athena or BigQuery for ad-hoc queries
- dbt for data transformations
- Airflow for orchestration

10. Monitoring & Alerting:

- Track ingestion lag
- Monitor queue depth
- Alert on anomalies in metrics
- Dashboard for system health

Coding and Debugging

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

1. Write a Django custom management command that accepts a file path and imports user data from a CSV file. Handle potential errors gracefully.

Solution

Create a file in **management/commands/import_users.py**:

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

class Command(BaseCommand):
    def add_arguments(self, parser):
        parser.add_argument('file_path', type=str)

    def handle(self, *args, **options):
        try:
            with open(options['file_path'], 'r') as f:
                for row in csv.DictReader(f):
                    User.objects.get_or_create(email=row['email'], defaults={'name': row['name']})
            self.stdout.write(self.style.SUCCESS('Import completed'))
        except Exception as e:
            self.stderr.write(self.style.ERROR(f'Error: {str(e)}'))
```

Key points:

- Use BaseCommand for custom commands
- add_arguments() defines CLI parameters
- get_or_create() prevents duplicates
- Proper exception handling with styled output

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

Problem Example

```
# N+1 Query Problem
for author in Author.objects.all():
    print(author.books.count())  # Hits DB each time
```

Solutions

1. Using select_related (ForeignKey/OneToOne):

```
books = Book.objects.select_related('author').all()
```

2. Using prefetch_related (ManyToMany/Reverse FK):

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

Debugging Tools:

- **django-debug-toolbar:** Visual query inspection
- **connection.queries:** Print all queries in development
- **logging:** Enable SQL logging in settings
- **nplusone:** Third-party package for detection

3. Implement a Django middleware that logs the execution time of each request and identifies slow endpoints (>500ms).

Custom Timing Middleware

```
import time
import logging

logger = logging.getLogger(__name__)

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

    def __call__(self, request):
        start = time.time()
        response = self.get_response(request)
        duration = (time.time() - start) * 1000
        if duration > 500:
            logger.warning(f'Slow request: {request.path} took {duration:.2f}ms')
        return response
```

Add to settings.py MIDDLEWARE:

```
MIDDLEWARE = [
    'myapp.middleware.RequestTimingMiddleware',
    # ... other middleware
]
```

Key concepts:

- Middleware processes requests/responses globally
- Use time.time() for performance measurement
- Logger integration for monitoring

4. What is monkey patching in Django? Provide an example where you might use it and explain the risks.

Monkey Patching Definition

Monkey patching is dynamically modifying a class or module at runtime to change its behavior without altering the source code.

Example Use Case

```
# In apps.py or __init__.py
from django.contrib.auth.models import User

def get_full_name_custom(self):
    return f'{self.first_name} {self.last_name}'.strip() or self.username

User.add_to_class('get_full_name', get_full_name_custom)
```

Legitimate Use Cases

- Extending third-party packages without forking
- Hot-fixing critical bugs in production
- Adding temporary debugging hooks
- Testing and mocking

Risks and Best Practices

- **Maintenance nightmare:** Hard to track changes
- **Update conflicts:** May break with library updates
- **Better alternatives:** Use model inheritance, proxy models, or custom managers
- **Documentation:** Always document monkey patches clearly

5. How do you profile memory usage in a Django application? Write code to identify

memory leaks in a view.

Memory Profiling Techniques

1. Using memory_profiler:

```
from memory_profiler import profile

@profile
def my_view(request):
    large_data = Model.objects.all()
    processed = [item.process() for item in large_data]
    return JsonResponse({'count': len(processed)})
```

2. Using tracemalloc (built-in):

```
import tracemalloc

def debug_view(request):
    tracemalloc.start()
    # Your code here
    result = expensive_operation()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f'Current: {current/1024/1024:.2f}MB, Peak: {peak/1024/1024:.2f}MB')
    return HttpResponse(result)
```

Common Memory Leak Causes

- **Unbounded querysets:** Use iterator() for large datasets
- **Circular references:** Especially with signals
- **Global caches:** Without proper eviction
- **File handles:** Not properly closed

Tools: django-silk, py-spy, objgraph

6. Implement a Django signal handler that prevents infinite recursion when updating the same model instance.

Problem: Signal Recursion

When a post_save signal modifies and saves the same instance, it triggers itself infinitely.

Solution with Flag Pattern

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

@receiver(post_save, sender=MyModel)
def update_related(sender, instance, created, **kwargs):
    if hasattr(instance, '_signal_processing'):
        return

    instance._signal_processing = True
    instance.calculated_field = instance.compute_value()
    instance.save(update_fields=['calculated_field'])
    del instance._signal_processing
```

Alternative: Disconnect/Reconnect

```
@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    post_save.disconnect(my_handler, sender=sender)
    instance.field = 'updated'
    instance.save()
    post_save.connect(my_handler, sender=sender)
```

Best practice: Use update_fields to limit what triggers signals

7. Write a custom Django template filter that safely truncates HTML content while preserving tag integrity.

Custom Template Filter

```
from django import template
from django.utils.html import strip_tags
from django.utils.text import Truncator

register = template.Library()

@register.filter(name='smart_truncate')
def smart_truncate(value, length=100):
    truncator = Truncator(value)
    return truncator.chars(length, html=True)
```

Usage in Template

```
{{ article.content|smart_truncate:150 }}
```

Advanced Version with Custom Logic

```
from html.parser import HTMLParser

@register.filter
def truncate_html(text, length):
    if len(strip_tags(text)) <= length:
        return text
    truncated = Truncator(text).chars(length, html=True, truncate=' ...')
    return truncated
```

Key features:

- Preserves HTML tags when truncating
- Counts only visible characters
- Properly closes open tags
- Django's Truncator handles edge cases

8. How would you handle database connection pooling in Django? Provide configuration for high-traffic scenarios.

Database Connection Pooling

Django doesn't include built-in connection pooling. For high-traffic apps, use **django-db-pool** or **pgbouncer**.

Using django-db-pool

```
DATABASES = {
    'default': {
        'ENGINE': 'django_db_pool.backends.postgresql',
        'NAME': 'mydb',
        'POOL_OPTIONS': {
            'POOL_SIZE': 10,
            'MAX_OVERFLOW': 20,
            'RECYCLE': 3600,
            'TIMEOUT': 30
        }
    }
}
```

Using Persistent Connections (Built-in)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'CONN_MAX_AGE': 600,  # 10 minutes
        'OPTIONS': {
```

```
        'connect_timeout': 10,
      }
   }
}
```

Best practices:

- Set CONN_MAX_AGE for persistent connections
- Use pgbouncer for PostgreSQL in production
- Monitor connection usage with database tools
- Adjust pool size based on worker count

9. Debug this Django ORM query that's producing incorrect results due to annotation ordering. Explain the issue and fix it.

Problematic Code

```
# Bug: Annotation after filter affects grouping
Product.objects.filter(category='electronics')
   .annotate(total_sales=Sum('order__quantity'))
   .filter(total_sales__gt=100)
```

The Problem

When you **annotate after filtering**, the annotation may include unwanted grouped data or produce unexpected NULL values.

Correct Approach

```
# Fix: Use proper ordering and conditional aggregation
from django.db.models import Q, Sum

Product.objects.annotate(
   total_sales=Sum('order__quantity',
                   filter=Q(category='electronics'))
).filter(total_sales__gt=100)
```

Alternative Solution

```
# Using subquery for complex cases
from django.db.models import OuterRef, Subquery

subq = Order.objects.filter(product=OuterRef('pk')).values('product')\
   .annotate(total=Sum('quantity')).values('total')
Product.objects.annotate(total_sales=Subquery(subq))\
   .filter(total_sales__gt=100)
```

Key lesson: Filter parameter in aggregation (Django 2.0+) prevents grouping issues

10. Implement exception handling in Django REST Framework that logs errors to an external service and returns user-friendly messages.

Custom Exception Handler

```
from rest_framework.views import exception_handler
import logging

logger = logging.getLogger(__name__)

def custom_exception_handler(exc, context):
   response = exception_handler(exc, context)

   if response is not None:
      logger.error(f'API Error: {exc}', extra={'context': context})
      response.data = {
         'error': str(exc),
         'status_code': response.status_code,
         'path': context['request'].path
```

```
        }
    return response
```

Configure in settings.py

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'myapp.utils.custom_exception_handler'
}
```

Integration with External Service

```
import sentry_sdk

def custom_exception_handler(exc, context):
    response = exception_handler(exc, context)
    if response is None:
        sentry_sdk.capture_exception(exc)
    return response
```

Benefits:

- Centralized error handling
- Consistent API error responses
- External monitoring integration
- User-friendly error messages

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 approach did you take?

Situation: Our e-commerce platform was experiencing 5+ second page load times during peak traffic, causing cart abandonments.

Task: I was tasked with reducing response times to under 1 second while maintaining functionality.

Action: I implemented a multi-layered approach: added database indexing on frequently queried fields, used select_related() and prefetch_related() to eliminate N+1 queries, implemented Redis caching for product listings, and added database query logging with Django Debug Toolbar to identify bottlenecks.

Result: Page load times dropped to 800ms on average, cart abandonment decreased by 23%, and we could handle 3x more concurrent users without scaling infrastructure.

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, function-based views, and no test coverage that needed upgrading to Django 3.2.

Task: Modernize the codebase while ensuring zero downtime and no regression bugs.

Action: I created a comprehensive test suite achieving 85% coverage before making changes, migrated function-based views to class-based views incrementally, replaced deprecated middleware, updated third-party packages, and performed the Django upgrade in stages (1.8 → 1.11 → 2.2 → 3.2) with thorough testing at each step.

Result: Successfully migrated with zero production incidents, improved code maintainability by 60% (measured by cyclomatic complexity), and reduced technical debt significantly. The team velocity increased by 40% for new feature development.

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

Situation: A security audit revealed our Django REST API was vulnerable to mass assignment attacks, allowing users to modify fields they shouldn't have access to.

Task: Fix the vulnerability immediately without breaking existing API contracts for mobile clients.

Action: I implemented explicit field declarations in all serializers, added read_only_fields for sensitive attributes, created custom permission classes for fine-grained access control, conducted a code review of all serializers, and implemented automated security testing in our CI/CD pipeline using tools like Bandit and Safety.

Result: Patched the vulnerability within 48 hours, prevented potential data breaches, and established security-first coding standards. No API breaking changes were required, and we caught 12 additional security issues through the new automated checks.

4. Tell me about a time when you had to make a difficult architectural decision in a Django project. What factors did you consider?

Situation: Our monolithic Django application was becoming difficult to scale, and different teams were blocked waiting for deployments.

Task: Decide between continuing with the monolith, transitioning to microservices, or adopting a modular monolith approach.

Action: I conducted a thorough analysis comparing deployment complexity, team expertise, operational overhead, and business requirements. I created a proof-of-concept for a modular monolith using Django apps with clear boundaries, documented trade-offs, and presented findings to stakeholders. We chose the modular monolith as it provided better separation without microservices complexity.

Result: Teams could work independently with 70% fewer merge conflicts, deployment frequency increased from weekly to daily, and we maintained operational simplicity. The architecture allowed future migration to microservices if needed.

5. Describe a situation where you had to debug a complex production issue in Django. What was your methodology?

Situation: Users reported intermittent 500 errors in production that couldn't be reproduced in staging, affecting approximately 5% of requests randomly.

Task: Identify and resolve the issue with minimal information and no reliable reproduction steps.

Action: I enhanced logging with request IDs for tracing, implemented Sentry for better error tracking with context, analyzed database slow query logs, discovered a race condition in our caching layer where cache invalidation and updates weren't atomic, and implemented distributed locking using Redis to ensure cache consistency across multiple application servers.

Result: Eliminated the 500 errors completely, improved system reliability from 95% to 99.9%, and established better observability practices that helped prevent future issues. The debugging methodology became our team standard.

6. Tell me about a time when you had to mentor junior developers on Django best practices. How did you approach it?

Situation: Three junior developers joined our team with limited Django experience, and code reviews were revealing repeated anti-patterns like N+1 queries and improper use of ORM.

Task: Bring them up to speed quickly while maintaining code quality and not creating bottlenecks in our development process.

Action: I created a Django best practices guide with code examples, conducted weekly knowledge-sharing sessions on topics like ORM optimization, security, and testing, implemented pair programming for complex features, set up pre-commit hooks to catch common issues early, and provided constructive code reviews with explanations rather than just corrections.

Result: Within three months, junior developers were contributing independently with 90% fewer code review iterations, two of them presented internal tech talks, and the best practices guide became part of our onboarding documentation. Team productivity increased by 35%.

7. Share an experience where you had to integrate a third-party service with Django and encountered unexpected challenges.

Situation: We needed to integrate a payment gateway with strict PCI compliance requirements, but their API had poor documentation and unpredictable rate limiting.

Task: Implement a reliable payment integration that handles failures gracefully and maintains compliance.

Action: I implemented an idempotent payment processing system using Django's transaction management, created a retry mechanism with exponential backoff using Celery, implemented webhook handling with signature verification for payment confirmations, added comprehensive logging without exposing sensitive data, and built a circuit breaker pattern to handle API unavailability.

Result: Achieved 99.95% payment success rate, zero compliance issues during audit, and the integration handled Black Friday traffic (10x normal) without failures. The pattern I established was reused for five other third-party integrations.

8. Describe a time when you disagreed with a technical decision made by your team regarding Django implementation. How did you handle it?

Situation: The team decided to use Django signals extensively for business logic, which I believed

would create hard-to-debug implicit dependencies and tight coupling.

Task: Address my concerns without undermining team decisions or creating conflict.

Action: I prepared a technical presentation showing concrete examples of signal-related issues I'd experienced, demonstrated alternative approaches using service layers and explicit method calls, created a proof-of-concept comparing both approaches with metrics on testability and maintainability, and facilitated a team discussion focusing on trade-offs rather than absolutes.

Result: The team agreed to limit signals to framework-level concerns (like cache invalidation) and use explicit service methods for business logic. This decision prevented debugging nightmares six months later when requirements became more complex. The collaborative approach strengthened team trust.

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

Situation: Our Django application had accumulated significant technical debt with outdated dependencies and poor test coverage, but business was pressuring for new features to meet a competitive threat.

Task: Deliver critical features while preventing technical debt from becoming unmanageable.

Action: I quantified the technical debt impact with metrics (deployment time, bug rate, development velocity), negotiated a 70/30 split (70% features, 30% tech debt) with stakeholders by showing how debt was slowing feature delivery, implemented the 'Boy Scout Rule' (leave code better than you found it), prioritized debt that blocked feature work, and automated dependency updates with Dependabot.

Result: Delivered all critical features on time, reduced technical debt by 40% over six months, and improved deployment confidence. Bug rate decreased by 50%, and the business saw the value in maintaining code health. The 70/30 approach became company policy.

10. Describe a situation where you had to improve the testing strategy for a Django application. What changes did you implement?

Situation: Our Django project had 30% test coverage, tests took 45 minutes to run, and developers were skipping them locally, leading to frequent production bugs.

Task: Improve test coverage and make testing a natural part of the development workflow.

Action: I implemented a testing pyramid strategy with unit tests for business logic, integration tests for API endpoints using Django REST framework's APITestCase, used factory_boy for test data generation instead of fixtures, parallelized test execution with pytest-xdist reducing runtime to 8 minutes, added coverage requirements to CI/CD blocking merges below 80%, and created testing guidelines with examples.

Result: Test coverage increased to 87%, production bugs decreased by 65%, developers started running tests locally due to faster feedback, and the team's confidence in refactoring improved significantly. Test-driven development became the norm rather than the exception.