Django Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. Create a custom model field that validates and stores JSON data with schema validation

Solution:

```
class JSONSchemaField(models.JSONField):
    def __init__(self, schema, *args, **kwargs):
        self.schema = schema
        super().__init__(*args, **kwargs)

    def validate(self, value, model_instance):
        super().validate(value, model_instance)
        jsonschema.validate(value, self.schema)
```

2. Implement a custom middleware that logs all database queries with their execution time for requests taking longer than 1 second

Solution:

```
from django.db import connection
from time import time

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

    def __call__(self, request):
        start_time = time()
        response = self.get_response(request)
        if time() - start_time > 1:
            print([q['sql'] for q in connection.queries])
```

3. Create a custom model manager that automatically filters soft-deleted records

Solution:

```
class ActiveManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(is_deleted=False)

class MyModel(models.Model):
    is_deleted = models.BooleanField(default=False)
    objects = ActiveManager()
```

4. Implement a decorator that caches expensive view results but invalidates cache when related models change

Solution:

```
from django.core.cache import cache
from functools import wraps

def smart_cache(timeout=300):
    def decorator(view_func):
        @wraps(view_func)
```

```
        def wrapper(request, *args, **kwargs):
            cache_key = f'view_{request.path}_{request.user.id}'
            result = cache.get(cache_key)
            return result if result else cache.set(cache_key, view_func(request, *args, **kwargs), timeout)
        return wrapper
    return decorator
```

5. Create a custom template filter that formats large numbers with K/M/B suffixes

Solution:

```
from django import template

register = template.Library()

@register.filter
def format_number(value):
    if value >= 1000000000: return f'{value/1000000000:.1f}B'
    if value >= 1000000: return f'{value/1000000:.1f}M'
    if value >= 1000: return f'{value/1000:.1f}K'
    return str(value)
```

6. Implement a custom field that automatically encrypts data before saving and decrypts when retrieving

Solution:

```
from django.db import models
from cryptography.fernet import Fernet

class EncryptedField(models.CharField):
    def __init__(self, *args, **kwargs):
        self.key = Fernet.generate_key()
        self.fernet = Fernet(self.key)
        super().__init__(*args, **kwargs)

    def from_db_value(self, value, *args):
        return self.fernet.decrypt(value.encode()).decode() if value else value
```

7. Create a signal handler that maintains a denormalized count of related objects

Solution:

```
@receiver(post_save, sender=Comment)
def update_comment_count(sender, instance, created, **kwargs):
    if created:
        Post.objects.filter(id=instance.post_id).update(
            comment_count=F('comment_count') + 1
        )
```

8. Implement a custom admin action that generates and emails reports asynchronously

Solution:

```
@admin.action(description='Generate report')
def generate_report(modeladmin, request, queryset):
    transaction.on_commit(lambda: async_task.delay(
        queryset.values_list('id', flat=True)
    ))
    messages.success(request, 'Report generation started')
```

9. Create a custom authentication backend that validates against an external API

Solution:

```
class ExternalAPIAuth(BaseBackend):
    def authenticate(self, request, username=None, password=None):
```

```
        response = requests.post('api/auth', json={'username': username, 'password': password})
        if response.status_code == 200:
            return User.objects.get_or_create(username=username)[0]
        return None
```

10. Implement a custom migration operation that populates data based on complex business logic

Solution:

```
class PopulateDataOperation(migrations.RunPython):
    def __init__(self):
        super().__init__(self.populate_data)

    def populate_data(apps, schema_editor):
        Model = apps.get_model('myapp', 'MyModel')
        Model.objects.bulk_create([Model(field=compute_value(i)) for i in range(100)])
```

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 Cache in Django with a size limit of N items?

LRU Cache Implementation

We can implement an LRU Cache using OrderedDict from collections:

```
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)
```

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

2. Implement a function to find all pairs of numbers in a list that sum to a target value using Django's built-in data structures.

Pair Sum Solution

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

Time Complexity: O(n), **Space Complexity:** O(n)

3. How would you implement a sliding window algorithm to find the maximum sum subarray of size k in Django?

Sliding Window Implementation

```
def max_sum_subarray(arr, k):
    if not arr or k > len(arr): return 0
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(len(arr) - k):
        window_sum = window_sum - arr[i] + arr[i + k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

4. Design a rate limiter using Django's cache framework. How would you implement token bucket algorithm?

Rate Limiter Implementation

```
from django.core.cache import cache
from time import time
```

```
def check_rate_limit(user_id, limit=100, window=3600):
    key = f'rate_limit:{user_id}'
    current = cache.get(key, {'count': 0, 'window': time()})
    if time() - current['window'] >= window:
        current = {'count': 1, 'window': time()}
```

5. Implement a custom priority queue for Django background tasks using heapq.

Priority Queue Implementation

```
import heapq

class TaskQueue:
    def __init__(self):
        self._queue = []
        self._index = 0
    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1
```

Time Complexity: O(log n) for push and pop operations

6. How would you implement a trie data structure for efficient prefix searching in Django?

Trie Implementation

```
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
```

Use Cases: Autocomplete, prefix matching, dictionary implementations

7. Implement an efficient method to detect cycles in a Django model's foreign key relationships.

Cycle Detection in Models

```
def detect_cycles(model, visited=None, rec_stack=None):
    if visited is None: visited, rec_stack = set(), set()
    if model in rec_stack: return True
    if model in visited: return False
    visited.add(model)
    rec_stack.add(model)
    return False
```

Algorithm: Uses DFS with recursion stack tracking

8. Design a custom cache invalidation strategy using Django signals and Redis sorted sets.

Cache Invalidation Strategy

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

@receiver(post_save)
def invalidate_cache(sender, instance, **kwargs):
    redis_client.zadd('cache_keys',
                      {f'{sender.__name__}:{instance.id}': time()})
    redis_client.expire('cache_keys', 3600)
```

9. Implement a consistent hashing algorithm for Django's cache backend distribution.

Consistent Hashing Implementation

```
import hashlib

def get_node(key, nodes, replicas=100):
    hash_key = int(hashlib.md5(str(key).encode()).hexdigest(), 16)
    node_point = hash_key % (replicas * len(nodes))
    return nodes[node_point // replicas]
```

Benefits: Minimizes cache misses during node addition/removal

10. Design an efficient algorithm for batch processing Django model instances with memory constraints.

Batch Processing Implementation

```
from django.db import transaction

def batch_process(queryset, batch_size=1000):
    start = 0
    while True:
        with transaction.atomic():
            batch = list(queryset[start:start + batch_size])
            if not batch: break
            process_batch(batch)
```

Memory Usage: O(batch_size)

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

Key Components:

- **Data Model:** URL table with original URL, short code, created date
- **Hash Generation:** MD5/Base62 encoding for unique short codes
- **Caching Layer:** Redis for frequently accessed URLs
- **Load Balancer:** Nginx for distributing traffic

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

Scalability Considerations:

- Implement read replicas for DB scaling
- Use CDN for global distribution
- Rate limiting to prevent abuse

2. How would you design a real-time chat system using Django Channels?

Architecture Components:

- **WebSocket Handler:** Django Channels for real-time communication
- **Message Broker:** Redis for pub/sub functionality
- **Storage:** PostgreSQL for message persistence

```
class ChatConsumer(WebsocketConsumer):
    async def connect(self):
        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )
```

Scaling Considerations:

- Multiple Channel layers for horizontal scaling
- Message queue for handling high load
- Proper connection pooling

3. Design a social media feed system with Django

Core Components:

- **Feed Generation:** Fanout-on-write vs Fanout-on-read
- **Caching Strategy:** Redis for feed caching
- **Database:** PostgreSQL with proper indexing

```
class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    likes = models.ManyToManyField(User, related_name='liked_posts')
```

Performance Optimizations:

- Implement cursor-based pagination
- Denormalization for faster reads
- Background task processing with Celery

4. Design a distributed task queue system using Django and Celery

System Components:

- **Message Broker:** RabbitMQ/Redis
- **Task Workers:** Celery workers
- **Result Backend:** Redis/PostgreSQL

```
@shared_task(bind=True, retry_backoff=True)
def process_large_dataset(self, dataset_id):
    try:
        return DatasetProcessor.process(dataset_id)
    except Exception as exc:
        self.retry(exc=exc)
```

Scaling Strategy:

- Multiple worker pools
- Task prioritization
- Proper error handling and retries

5. How would you implement a scalable search functionality in Django?

Architecture:

- **Search Engine:** Elasticsearch/PostgreSQL Full-Text Search
- **Index Management:** Automatic indexing via signals
- **Caching:** Redis for search results

```
class SearchIndex(models.Model):
    content = models.TextField()
    vector = SearchVectorField()
    created_at = models.DateTimeField(auto_now_add=True)
    index_trigger = models.signals.post_save
```

Optimization Techniques:

- Implement faceted search
- Use search result highlighting
- Implement autocomplete using trigrams

6. Design a notification system for a large-scale Django application

Key Components:

- **Notification Types:** Email, Push, In-app
- **Queue System:** Celery for async processing
- **Storage:** PostgreSQL + Redis

```
class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    type = models.CharField(max_length=20)
    message = models.TextField()
    is_read = models.BooleanField(default=False)
```

Scaling Considerations:

- Batch processing for bulk notifications
- Template-based notification system
- Rate limiting and throttling

7. Design a caching strategy for a high-traffic Django application

Caching Layers:

- **Page Cache:** Varnish/Nginx
- **Application Cache:** Redis/Memcached
- **Database Cache:** PostgreSQL query cache

```
@cache_page(60 * 15)
@vary_on_cookie
def view_product(request, product_id):
    product = Product.objects.select_related('category')
                        .get(id=product_id)
```

Implementation Strategy:

- Cache invalidation patterns
- Cache warming strategies
- Multi-tier caching architecture

8. Design an API rate limiting system in Django

Components:

- **Rate Limiter:** Token bucket/Leaky bucket algorithm
- **Storage:** Redis for rate limit counters
- **Middleware:** Custom rate limiting middleware

```
class RateLimitMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        self.cache = RedisCache()
        self.rate_limit = 100  # requests per minute
```

Features:

- Different limits for different user tiers
- Burst handling
- Custom response headers

9. Design a file upload system for large files in Django

System Components:

- **Storage:** S3/Cloud Storage
- **Upload:** Chunked upload with resumability
- **Processing:** Async task queue for post-processing

```
class FileUpload(models.Model):
    file = models.FileField(storage=S3Boto3Storage())
    chunks_received = models.IntegerField(default=0)
    upload_status = models.CharField(max_length=20)
```

Features:

- Progress tracking
- Validation and virus scanning
- Automatic file type detection

10. Design a microservices architecture using Django

Architecture Components:

- **Service Discovery:** Consul/Eureka
- **API Gateway:** Kong/Nginx
- **Communication:** REST/gRPC

```
class UserService(models.Model):
    @api_view(['POST'])
    def create_user(request):
```

```
    serializer = UserSerializer(data=request.data)
    if serializer.is_valid():
```

Design Considerations:

- Service boundaries definition
- Data consistency patterns
- Circuit breaker implementation

Coding and Debugging

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

1. Write a Django middleware to log request processing time for each view

Solution:

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

    def __call__(self, request):
        import time
        start_time = time.time()
        response = self.get_response(request)
        duration = time.time() - start_time
        print(f'{request.path}: {duration:.2f}s')
```

Key points:

- Middleware must implement __init__ and __call__
- Captures timestamp before and after request processing
- Can be extended to log to file/database

2. Create a Django decorator to cache expensive view results with a timeout

Solution:

```
from functools import wraps
from django.core.cache import cache

def cache_result(timeout=300):
    def decorator(view_func):
        @wraps(view_func)
        def wrapper(request, *args, **kwargs):
            key = f'cache_{request.path}'
            result = cache.get(key)
            if result is None:
                result = view_func(request, *args, **kwargs)
                cache.set(key, result, timeout)
            return result
        return wrapper
    return decorator
```

3. Implement a custom Django model field that automatically encrypts data before saving

Solution:

```
from django.db import models
from cryptography.fernet import Fernet

class EncryptedField(models.TextField):
    def __init__(self, *args, **kwargs):
        self.key = Fernet.generate_key()
        self.fernet = Fernet(self.key)
        super().__init__(*args, **kwargs)

    def from_db_value(self, value, *args):
        return self.fernet.decrypt(value.encode()).decode()
```

```
    def to_python(self, value):
        return self.fernet.encrypt(str(value).encode())
```

4. Write a function to flatten a deeply nested QuerySet of related objects

Solution:

```
def flatten_queryset(queryset, related_fields):
    flat_data = []
    for obj in queryset.select_related(*related_fields):
        item = {}
        for field in related_fields:
            related_obj = obj
            for attr in field.split('__'):
                related_obj = getattr(related_obj, attr)
            item[field] = related_obj
        flat_data.append(item)
    return flat_data
```

5. Create a Django management command to cleanup stale sessions and expired tokens

Solution:

```
from django.core.management.base import BaseCommand
from django.contrib.sessions.models import Session
from django.utils import timezone

class Command(BaseCommand):
    def handle(self, *args, **kwargs):
        expired = Session.objects.filter(expire_date__lt=timezone.now())
        count = expired.count()
        expired.delete()
        self.stdout.write(f'Deleted {count} expired sessions')
```

6. Implement a custom Django authentication backend using JWT tokens

Solution:

```
from django.contrib.auth.backends import BaseBackend
import jwt
from users.models import User

class JWTBackend(BaseBackend):
    def authenticate(self, request, token=None):
        try:
            payload = jwt.decode(token, 'secret', algorithms=['HS256'])
            return User.objects.get(id=payload['user_id'])
        except:
            return None
```

Note:

- Requires proper secret key management
- Should include token expiration handling
- Consider adding refresh token logic

7. Write a function to bulk upsert Django model instances efficiently

Solution:

```
from django.db import transaction

def bulk_upsert(model, objects, key_fields):
    with transaction.atomic():
        existing = {
            tuple(getattr(obj, f) for f in key_fields): obj
            for obj in model.objects.all()
        }
```

```
        to_update = []
        to_create = []
        for obj in objects:
            key = tuple(getattr(obj, f) for f in key_fields)
            if key in existing:
                to_update.append(obj)
            else:
                to_create.append(obj)
        model.objects.bulk_create(to_create)
        model.objects.bulk_update(to_update, fields=[f for f in obj.__dict__])
```

8. Create a Django signal to automatically generate slugs for model instances

Solution:

```
from django.db.models.signals import pre_save
from django.utils.text import slugify
from django.dispatch import receiver

@receiver(pre_save, sender=YourModel)
def generate_slug(sender, instance, **kwargs):
    if not instance.slug:
        base_slug = slugify(instance.title)
        slug = base_slug
        counter = 1
        while sender.objects.filter(slug=slug).exists():
            slug = f'{base_slug}-{counter}'
            counter += 1
        instance.slug = slug
```

9. Implement a custom Django template filter to format large numbers

Solution:

```
from django import template
register = template.Library()

@register.filter
def format_number(value):
    try:
        value = float(value)
        if value >= 1000000:
            return f'{value/1000000:.1f}M'
        elif value >= 1000:
            return f'{value/1000:.1f}K'
        return str(value)
    except (ValueError, TypeError):
        return value
```

10. Write a Django view decorator to rate limit requests based on IP address

Solution:

```
from functools import wraps
from django.core.cache import cache
from django.http import HttpResponseTooManyRequests

def rate_limit(requests=100, window=3600):
    def decorator(view_func):
        @wraps(view_func)
        def wrapper(request, *args, **kwargs):
            ip = request.META.get('REMOTE_ADDR')
            key = f'ratelimit_{ip}'
            count = cache.get(key, 0)
            if count >= requests:
                return HttpResponseTooManyRequests()
            cache.set(key, count + 1, window)
            return view_func(request, *args, **kwargs)
```

```
    return wrapper
return decorator
```