VibeVault Interview Script - Part 2

Complete deep-dive for all additional topics with: **Interviewer Question → Your Answer → Code with Line Comments → Why I Did This**

DJANGO TOPICS

---

1. CBV (Class Based Views)

Interviewer: "What are Class Based Views?" **You:** "CBVs are views written as Python classes instead of functions. They provide inheritance, mixins, and reusable patterns."

In my project: `backend/users/views.py`

```
# Line 33: I define a ViewSet which is a specialized CBV
class RegisterView(viewsets.ViewSet):
    # Line 38: I set permissions at the class level
    permission_classes = (AllowAny,)

    # Line 40: Methods handle different actions
    @action(detail=False, methods=['post'])
    def register(self, request):
        serializer = RegisterSerializer(data=request.data)
        if serializer.is_valid():
            user = serializer.save()
            return Response({'message': 'User registered'}, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Why I did this: "CBVs let me organize related logic in one class. I can set class-level attributes like `permission_classes` that apply to all methods."

---

2. Generic CBV

Interviewer: "What are Generic Class Based Views?" **You:** "Generic CBVs are pre-built views for common patterns like displaying lists, details, or handling forms."

In my project: *(Django's built-in generic views - concept explanation)*

```
# Example of Django's generic CBV pattern (similar to what DRF uses):
from django.views.generic import ListView, DetailView

class MemoryListView(ListView):
    model = Memory
    template_name = 'memories/list.html'
    context_object_name = 'memories'
```

```
    def get_queryset(self):
        return Memory.objects.filter(user=self.request.user)
```

Why I did this: "Generic views eliminate boilerplate. `ListView` handles pagination, queryset fetching, and template rendering automatically."

---

3. Sessions

Interviewer: "How does Django handle sessions?" **You:** "Django stores session data server-side and gives the client a session ID cookie. I have session middleware enabled."

In my project: `backend/config/settings.py`

```
# Line 37-40: Session-related apps are installed
INSTALLED_APPS = [
    'django.contrib.sessions',  # Handles session storage
    'django.contrib.auth',      # Uses sessions for login
]

# Line 62: Session middleware processes every request
MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
]
```

Why I did this: "Although I primarily use JWT for API auth, sessions are still needed for Django Admin login and CSRF protection."

---

4. Authentication (Django)

Interviewer: "How did you set up Django authentication?" **You:** "I extended `AbstractUser` for a custom User model and configured password validators."

In my project: `backend/users/models.py`

```
# Line 6: I import AbstractUser as my base
from django.contrib.auth.models import AbstractUser

# Line 9: My custom User inherits all auth functionality
class User(AbstractUser):
    bio = models.TextField(blank=True, null=True)
    avatar = models.ImageField(upload_to='avatars/', blank=True, null=True)

    def __str__(self):
        return f'{self.username} ({self.email})'
```

Why I did this: "AbstractUser gives me username, email, password, is_active, is_staff, etc. I just added `bio` and `avatar` for my app's needs."

---

5. APIView

Interviewer: "What is APIView in DRF?" **You:** "APIView is DRF's base class for all views. It adds authentication, permissions, and content negotiation to Django's View."

In my project: *(Implicit through ViewSets which inherit from APIView)*

```
# DRF's APIView is the foundation. Here's the concept:
from rest_framework.views import APIView
from rest_framework.response import Response

class HealthCheckView(APIView):
    permission_classes = []  # No auth needed

    def get(self, request):
        return Response({'status': 'healthy'})
```

Why I did this: "APIView provides `request.data` (parsed body), `request.user` (authenticated user), and automatic content type handling."

---

6. GenericAPIView

Interviewer: "What's the difference between APIView and GenericAPIView?" **You:** "GenericAPIView adds queryset handling, serializer integration, and pagination. It's between APIView and ViewSets."

In my project: *(Concept - ViewSets build on this)*

```
# GenericAPIView provides get_queryset(), get_serializer(), pagination
from rest_framework.generics import GenericAPIView

class MemoryListView(GenericAPIView):
    queryset = Memory.objects.all()
    serializer_class = MemorySerializer

    def get_queryset(self):
        # Automatically filtered by user
        return self.queryset.filter(user=self.request.user)
```

Why I did this: "GenericAPIView connects models and serializers. I override `get_queryset()` to filter data per user."

---

7. Mixins

Interviewer: "What are Mixins in DRF?" **You:** "Mixins provide reusable CRUD behaviors. You combine them with GenericAPIView to build custom endpoints."

In my project: *(Concept - used via ModelViewSet)*

```
# Mixins provide individual CRUD actions:
# - ListModelMixin     -> list()
# - CreateModelMixin   -> create()
# - RetrieveModelMixin -> retrieve()
# - UpdateModelMixin   -> update(), partial_update()
# - DestroyModelMixin  -> destroy()

# ModelViewSet combines ALL mixins:
class MemoryViewSet(viewsets.ModelViewSet):
    # I get list, create, retrieve, update, destroy automatically
    pass
```

Why I did this: "Mixins follow the DRY principle. Instead of writing CRUD logic repeatedly, I inherit it."

---

8. Generic Views (DRF)

Interviewer: "What are DRF's Generic Views?" **You:** "They combine GenericAPIView with mixins for common patterns like ListCreateAPIView."

In my project: *(Concept)*

```
# DRF provides pre-combined generic views:
from rest_framework.generics import ListCreateAPIView, RetrieveUpdateDestroyAPIView

class MemoryListCreate(ListCreateAPIView):
    # GET /memories/ -> list()
    # POST /memories/ -> create()
    queryset = Memory.objects.all()
    serializer_class = MemorySerializer

class MemoryDetail(RetrieveUpdateDestroyAPIView):
    # GET /memories/1/ -> retrieve()
    # PUT /memories/1/ -> update()
    # DELETE /memories/1/ -> destroy()
    queryset = Memory.objects.all()
    serializer_class = MemorySerializer
```

Why I did this: "Generic views reduce code. I don't write `def get()` or `def post()` - they're inherited."

---

9. ViewSets

Interviewer: "Why did you use ViewSets?" **You:** "ViewSets combine all CRUD actions in one class and work with Routers for automatic URL generation."

In my project: `backend/memories/views.py`

```
# Line 23: I inherit from ModelViewSet
class MemoryViewSet(viewsets.ModelViewSet):
```

```
    # Line 35: Class-level permission
    permission_classes = (IsAuthenticated,)

    # Line 36-40: Filtering and ordering
    filter_backends = (DjangoFilterBackend, filters.SearchFilter, filters.OrderingFilter)
    filterset_fields = ('analysis_status', 'media_type')
    search_fields = ('title', 'text')
    ordering_fields = ('created_at', 'updated_at')

    # Line 42-44: I filter to current user's data only
    def get_queryset(self):
        return Memory.objects.filter(user=self.request.user)
```

Why I did this: "One class handles 6 endpoints. Combined with Router, I get a full REST API in ~20 lines."

---

10. Filtering

Interviewer: "How do you filter API results?" **You:** "I use DjangoFilterBackend for exact matches and SearchFilter for text search."

In my project: `backend/memories/views.py`

```
# Line 9: I import filter backends
from django_filters.rest_framework import DjangoFilterBackend

# Line 36: I enable multiple filter backends
filter_backends = (DjangoFilterBackend, filters.SearchFilter, filters.OrderingFilter)

# Line 37: Exact match filters: ?analysis_status=done
filterset_fields = ('analysis_status', 'media_type')

# Line 38: Text search: ?search=vacation
search_fields = ('title', 'text')

# Line 39: Ordering: ?ordering=-created_at
ordering_fields = ('created_at', 'updated_at')
```

Why I did this: "Users can filter memories: `/memories/?media_type=photo&search=beach&ordering=-created_at`"

---

11. File Uploads (DRF)

Interviewer: "How do you handle file uploads in DRF?" **You:** "I use FileField in serializers with multipart/form-data content type."

In my project: `backend/memories/serializers.py`

```
# Line 55: Create serializer handles file uploads
class MemoryCreateSerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Memory
        fields = ('title', 'text', 'media_type', 'media_file')

    def create(self, validated_data):
        # File is automatically handled by DRF
        memory = Memory.objects.create(
            user=self.context['request'].user,
            **validated_data  # includes media_file
        )
        return memory
```

Why I did this: "DRF automatically parses multipart forms. The file is saved to `MEDIA_ROOT` via the model's FileField."

---

12. Versioning (DRF)

Interviewer: "How do you version your API?" **You:** "I use URL path versioning by prefixing all routes with `/api/v1/`."

In my project: `backend/config/urls.py`

```
# Line 12-13: Manual URL versioning
urlpatterns = [
    path('api/v1/auth/', include('users.urls')),     # Version 1
    path('api/v1/memories/', include('memories.urls')), # Version 1

    # Future: path('api/v2/memories/', include('memories.urls_v2')),
]
```

Why I did this: "Versioning lets me make breaking changes in v2 without breaking existing v1 clients."

---

13. Template Tags

Interviewer: "What are Template Tags?" **You:** "Template tags add logic to Django templates. I use built-in ones like `{% url %}` and `{% static %}`."

In my project: *(Used in Django Admin templates)*

```
<!-- Example of template tags -->
{% load static %}

<!DOCTYPE html>
<html>
<head>
    {% block title %}VibeVault{% endblock %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
```

```
    {% if user.is_authenticated %}
        <p>Welcome, {{ user.username }}</p>
    {% endif %}
</body>
</html>
```

Why I did this: "Template tags like `{% url 'login' %}` generate URLs dynamically. `{% static %}` resolves static file paths."

---

14. Filters (Template)

Interviewer: "What are Template Filters?" **You:** "Filters modify variables in templates using the pipe `|` syntax."

In my project: *(Concept)*

```
<!-- Filter examples -->
{{ memory.title|upper }}           <!-- VACATION PHOTOS -->
{{ memory.created_at|date:"F j, Y" }}  <!-- December 14, 2024 -->
{{ memory.text|truncatewords:20 }}     <!-- First 20 words... -->
{{ user.email|default:"No email" }}    <!-- Fallback value -->
```

Why I did this: "Filters format data for display without modifying the model. They keep templates clean."

---

15. base.html

Interviewer: "What is base.html?" **You:** "It's the parent template that defines the common structure. Child templates extend it."

In my project: *(Concept - used by Django Admin)*

```
<!-- base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}VibeVault{% endblock %}</title>
    {% block extra_css %}{% endblock %}
</head>
<body>
    <nav>{% block nav %}{% endblock %}</nav>
    <main>{% block content %}{% endblock %}</main>
    {% block extra_js %}{% endblock %}
</body>
</html>

<!-- child.html -->
{% extends "base.html" %}

{% block title %}Dashboard{% endblock %}
```

```
{% block content %}
    <h1>Welcome!</h1>
{% endblock %}
```

Why I did this: "Template inheritance avoids repeating HTML. I define `{% block %}` sections that child templates override."

---

16. Form

Interviewer: "Did you use Django Forms?" **You:** "For API work I use DRF Serializers, but Django Forms handle HTML form validation."

In my project: *(Concept)*

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)

    def clean_email(self):
        # Custom validation
        email = self.cleaned_data['email']
        if 'spam' in email:
            raise forms.ValidationError("Invalid email")
        return email
```

Why I did this: "Forms handle validation, error messages, and HTML rendering. `clean_<field>` methods add custom validation."

---

17. ModelForm

Interviewer: "What's the difference between Form and ModelForm?" **You:** "ModelForm generates fields from a Django model automatically."

In my project: *(Concept)*

```
from django import forms
from .models import Memory

class MemoryForm(forms.ModelForm):
    class Meta:
        model = Memory
        fields = ['title', 'text', 'media_type', 'media_file']
        widgets = {
            'text': forms.Textarea(attrs={'rows': 5}),
        }
```

```
    def save(self, user):
        memory = super().save(commit=False)
        memory.user = user
        memory.save()
        return memory
```

Why I did this: "ModelForm reads field types from the model. I don't need to define `CharField` for each field manually."

---

18. Create Migrations

Interviewer: "How do you create migrations?" **You:** "I run `makemigrations` after changing models. Django generates migration files automatically."

In my project: `backend/manage.py`

```
# Command to create migrations
python manage.py makemigrations

# Output:
# Migrations for 'memories':
#   memories/migrations/0001_initial.py
#     - Create model Memory
```

Migration file created:

```
# 0001_initial.py
class Migration(migrations.Migration):
    dependencies = [('users', '0001_initial')]

    operations = [
        migrations.CreateModel(
            name='Memory',
            fields=[
                ('id', models.BigAutoField(primary_key=True)),
                ('title', models.CharField(max_length=255)),
                # ... more fields
            ],
        ),
    ]
```

Why I did this: "Migrations track schema changes. They let me version control database changes and apply them consistently."

---

19. Apply Migrations

Interviewer: "How do you apply migrations to the database?" **You:** "I run `migrate` to execute pending migrations."

In my project: `backend/manage.py`

```
# Apply all pending migrations
python manage.py migrate

# Output:
# Operations to perform:
#   Apply all migrations: admin, auth, contenttypes, memories, sessions, users
# Running migrations:
#   Applying memories.0001_initial... OK

# Check migration status
python manage.py showmigrations
# [X] 0001_initial  <- Applied
# [ ] 0002_add_field <- Pending
```

Why I did this: "`migrate` creates/modifies database tables. It's safe to run multiple times - it only applies unapplied migrations."

---

20. Register Models (Admin)

Interviewer: "How do you add models to Django Admin?" **You:** "I use `@admin.register` decorator or `admin.site.register()`."

In my project: `backend/memories/admin.py`

```
# Line 4: I use the decorator syntax
@admin.register(Memory)
class MemoryAdmin(admin.ModelAdmin):
    list_display = ('id', 'user', 'title', 'analysis_status')

# Alternative syntax:
# admin.site.register(Memory, MemoryAdmin)
```

Why I did this: "Registering models makes them visible in `/admin/`. The decorator is cleaner than calling `register()` separately."

---

21. Display Options (Admin)

Interviewer: "How do you customize the Admin list view?" **You:** "I use `list_display`, `list_filter`, `search_fields`, and `ordering`."

In my project: `backend/memories/admin.py`

```
# Line 5-10: Admin customization options
@admin.register(Memory)
class MemoryAdmin(admin.ModelAdmin):
    # Columns shown in list view
```

```
    list_display = ('id', 'user', 'title', 'media_type', 'analysis_status', 'created_at')

    # Sidebar filters
    list_filter = ('media_type', 'analysis_status', 'created_at')

    # Search box searches these fields
    search_fields = ('title', 'text', 'user__username')

    # Fields that can't be edited
    readonly_fields = ('created_at', 'updated_at', 'metrics_json')

    # Default sort order
    ordering = ('-created_at',)
```

Why I did this: "Custom display options make data manageable. I can filter by status, search by title, and sort by date."

---

22. Custom Middleware

Interviewer: "Explain Custom Middleware." **You:** "I wrote `RequestTimingMiddleware` to measure and log API response times."

In my project: `backend/config/middleware.py`

```
# Line 11: My custom middleware class
class RequestTimingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # BEFORE the view
        start_time = time.time()

        # Call the view
        response = self.get_response(request)

        # AFTER the view
        duration = time.time() - start_time
        response['X-Process-Time'] = f"{duration:.4f}s"

        if duration > 1.0:
            logger.warning(f"Slow request: {request.path}")

        return response
```

Why I did this: "This lets me monitor every API call's latency without adding logging to each view."

---

23. Signals

Interviewer: "Explain Signals with an example." **You:** "I use `post_save` signal to create a welcome memory when a new user registers."

In my project: `backend/users/signals.py`

```
# Line 6: I import the signal
from django.db.models.signals import post_save
from django.dispatch import receiver

# Line 14: @receiver connects my function to the signal
@receiver(post_save, sender=User)
def create_welcome_memory(sender, instance, created, **kwargs):
    # Line 24: Only run for NEW users
    if created:
        Memory.objects.create(
            user=instance,
            title="Welcome to VibeVault! 🚀",
            text="This is your first memory...",
            media_type='text',
            analysis_status='done'
        )
```

Why I did this: "Signals decouple logic. The registration view doesn't know about memories - the signal handles that side effect."

---

24. Connection (Database)

Interviewer: "How do you configure database connection?" **You:** "I set the `DATABASES` dict in settings with connection parameters from environment variables."

In my project: `backend/config/settings.py`

```
# Line 93-102: Database configuration
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRES_DB', 'vibevault'),
        'USER': os.getenv('POSTGRES_USER', 'postgres'),
        'PASSWORD': os.getenv('POSTGRES_PASSWORD', 'postgres'),
        'HOST': os.getenv('POSTGRES_HOST', 'localhost'),
        'PORT': os.getenv('POSTGRES_PORT', '5432'),
    }
}
```

Why I did this: "I use environment variables so the same code works in dev (`localhost`) and prod (AWS RDS) without changes."

---

25. Serializer

Interviewer: "What is a Serializer?" **You:** "Serializers convert complex data (models, querysets) to Python primitives that can become JSON."

In my project: `backend/memories/serializers.py`

```
# Line 9: I inherit from ModelSerializer
class MemorySerializer(serializers.ModelSerializer):
    # Line 11: Custom field representation
    user = serializers.StringRelatedField(read_only=True)

    class Meta:
        model = Memory
        fields = ('id', 'user', 'title', 'text', 'created_at')
```

Why I did this: "Serializers handle both directions: Model→JSON (serialization) and JSON→Model (deserialization with validation)."

---

26. ModelSerializer

Interviewer: "What's the advantage of ModelSerializer?" **You:** "It generates fields from the model automatically and includes default validators."

In my project: `backend/memories/serializers.py`

```
class MemorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Memory
        # I list which fields to include
        fields = ('id', 'user', 'title', 'text', 'media_type', 'created_at')
        # These fields can't be modified by API requests
        read_only_fields = ('id', 'user', 'created_at')
```

Why I did this: "ModelSerializer inspects the model and creates appropriate field types. `max_length`, `required`, etc. come from the model."

---

27. Nested Serializers

Interviewer: "How do you serialize related data?" **You:** "I use SerializerMethodField to flatten or transform nested data."

In my project: `backend/memories/serializers.py`

```
class MemorySerializer(serializers.ModelSerializer):
    # Line 12-13: Computed fields
    emotion = serializers.SerializerMethodField()
    sentiment = serializers.SerializerMethodField()

    # Line 24-26: Method extracts from JSON field
    def get_emotion(self, obj):
```

```
        if obj.metrics_json and 'emotion' in obj.metrics_json:
            return obj.metrics_json['emotion'].get('top_emotion')
        return None
```

Why I did this: "The database stores complex JSON. SerializerMethodField lets me return just `emotion: 'joy'` instead of the full nested structure."

---

28. Built-in Permissions

Interviewer: "What permissions does DRF provide?" **You:** "DRF has `IsAuthenticated`, `IsAdminUser`, `AllowAny`, and `IsAuthenticatedOrReadOnly`."

In my project: `backend/config/settings.py` & `backend/users/views.py`

```
# settings.py - Global default
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',  # Require login globally
    ),
}

# views.py - Override per view
class RegisterView(viewsets.ViewSet):
    permission_classes = (AllowAny,)  # Anyone can register
```

Why I did this: "I default to `IsAuthenticated` for security. Public endpoints like register/login explicitly use `AllowAny`."

---

29. Custom Permission

Interviewer: "Did you write any custom permissions?" **You:** "The queryset filtering acts as object-level permission - users only see their own memories."

In my project: `backend/memories/views.py`

```
# Line 42-44: Implicit permission via queryset
def get_queryset(self):
    # Users can ONLY access their own memories
    return Memory.objects.filter(user=self.request.user)

# Explicit custom permission would look like:
# from rest_framework.permissions import BasePermission
#
# class IsOwner(BasePermission):
#     def has_object_permission(self, request, view, obj):
#         return obj.user == request.user
```

Why I did this: "Filtering the queryset is simpler for my use case. Users can't even see URLs for others' memories."

---

30. Exception Handling

Interviewer: "How does DRF handle errors?" **You:** "DRF has a default exception handler that converts exceptions to proper HTTP responses."

In my project: `backend/memories/views.py`

```
# Line 79-83: I use DRF's built-in error handling
@action(detail=False, methods=['get'])
def search(self, request):
    query = request.query_params.get('q', '')

    if not query:
        # DRF converts this to proper JSON error response
        return Response(
            {'error': 'Query parameter "q" is required'},
            status=status.HTTP_400_BAD_REQUEST
        )
```

Why I did this: "DRF automatically formats errors as JSON with proper status codes. I don't need try/except everywhere."

---

31. Django Tests

Interviewer: "How do you test Django code?" **You:** "I use Django's TestCase which provides database transactions and test client."

In my project: `backend/tests/` *(Testing concept)*

```
from django.test import TestCase
from django.contrib.auth import get_user_model

User = get_user_model()

class UserModelTest(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            username='testuser',
            email='test@example.com',
            password='testpass123'
        )

    def test_user_creation(self):
        self.assertEqual(self.user.username, 'testuser')
        self.assertTrue(self.user.check_password('testpass123'))

    def test_user_str(self):
        self.assertEqual(str(self.user), 'testuser (test@example.com)')
```

Why I did this: "TestCase wraps each test in a transaction that's rolled back. Tests are isolated and don't affect the real database."

---

32. DRF Tests

Interviewer: "How do you test API endpoints?" **You:** "I use DRF's APITestCase with APIClient for authenticated requests."

In my project: `backend/tests/` *(Testing concept)*

```
from rest_framework.test import APITestCase, APIClient
from rest_framework import status

class MemoryAPITest(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='test', password='pass')
        self.client = APIClient()
        self.client.force_authenticate(user=self.user)

    def test_list_memories(self):
        response = self.client.get('/api/v1/memories/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_create_memory(self):
        data = {'title': 'Test', 'text': 'Content', 'media_type': 'text'}
        response = self.client.post('/api/v1/memories/', data)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

Why I did this: " `force_authenticate` bypasses JWT for testing. I can test protected endpoints without token management."

---

33. Config

Interviewer: "How do you organize Django configuration?" **You:** "I have a `config` package with settings, URLs, and ASGI/WSGI applications."

In my project: `backend/config/` structure

```
config/
├── __init__.py      # Loads Celery
├── settings.py      # All Django settings
├── urls.py          # Main URL routing
├── wsgi.py          # Sync server entry
├── asgi.py          # Async server entry (Channels)
├── celery.py        # Celery configuration
└── middleware.py    # Custom middleware
```

Why I did this: "Keeping config in a dedicated package separates project configuration from application code (users, memories, etc.)."

---

34. Signals Load

Interviewer: "How do you ensure signals are loaded?" **You:** "I import the signals module in the app's `ready()` method."

In my project: `backend/users/apps.py`

```
# Line 8: App configuration class
class UsersConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'users'

    # Line 12: ready() is called when Django starts
    def ready(self):
        # Line 14: Import signals to register them
        import users.signals
```

Why I did this: "Without this import, the `@receiver` decorators never run. The signal handlers wouldn't be connected."

✅ Complete Summary - Part 2

All **34 additional topics** covered with:

- Interviewer question
- Your answer
- Line-by-line code
- Why I did this