VibeVault Complete Interview Script

Every topic and sub-topic explained with: **Interviewer Question → Your Answer → Code with Line Comments → Why I Did This**

DJANGO TOPICS

---

1. Project Commands

Interviewer: "What are Django Project Commands?" **You:** "These are CLI commands run via `manage.py`. Django has built-in ones like `runserver` and `migrate`, and I wrote a **custom one** to seed demo data."

In my project: `backend/manage.py`

```
# Line 1: Standard shebang for executable scripts
#!/usr/bin/env python

# Line 3: I import os to set environment variables
import os
import sys

# Line 5: I use dotenv to load secrets from .env file
from dotenv import load_dotenv
load_dotenv()

# Line 10: The main function that Django calls
def main():
    # Line 12: I tell Django which settings file to use
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

    # Line 14: I import Django's command runner
    from django.core.management import execute_from_command_line

    # Line 21: This runs whatever command was passed (runserver, migrate, etc.)
    execute_from_command_line(sys.argv)
```

Why I did this: "This is the entry point for all Django CLI operations. I added `load_dotenv()` so environment variables are available before Django initializes."

Sub-topic: BaseCommand

Interviewer: "How do you create a custom management command?" **You:** "I inherit from `BaseCommand` and implement the `handle()` method."

In my project: `backend/management/commands/seed_demo.py`

```python
# Line 6: I import BaseCommand - the parent class for all commands
from django.core.management.base import BaseCommand, CommandError

# Line 69: My command class MUST be named 'Command'
class Command(BaseCommand):
    # Line 70: This shows up when you run 'python manage.py help seed_demo'
    help = 'Seed the database with demo user and sample memories'
```

Why I did this: "Django discovers commands by looking for `Command` classes in `management/commands/` folders. The `help` attribute provides documentation."

Sub-topic: add_arguments

Interviewer: "Can custom commands accept command-line flags?" **You:** "Yes, I override `add_arguments()` to define flags like `--clear`."

In my project: `backend/management/commands/seed_demo.py`

```python
# Line 72: I define what arguments my command accepts
def add_arguments(self, parser):
    # Line 73: I add an optional --clear flag
    parser.add_argument(
        '--clear',
        action='store_true',  # This makes it a boolean flag
        help='Clear existing demo data before seeding'
    )
```

Why I did this: "This lets me run `python manage.py seed_demo --clear` to reset data. The `action='store_true'` means the flag doesn't need a value."

Sub-topic: handle()

Interviewer: "Where does the actual command logic go?" **You:** "In the `handle()` method. This is what Django calls when you run the command."

In my project: `backend/management/commands/seed_demo.py`

```python
# Line 79: The main logic of my command
def handle(self, *args, **options):
    # Line 80: I print a status message
    self.stdout.write(self.style.SUCCESS('Starting demo data seed...'))

    # Line 91: I check if the --clear flag was passed
    if options['clear']:
        Memory.objects.filter(user=user).delete()

    # Line 98: I create the demo user
    user = User.objects.create_user(
        username='demo',
```

```
        email='demo@vibevault.local',
        password='DemoPassword123!'
    )
```

Why I did this: "`handle()` receives the parsed arguments in `options`. I use `self.stdout.write` instead of `print()` so output works correctly in all environments."

Sub-topic: self.style.SUCCESS

Interviewer: "How do you color-code command output?" **You:** "I use `self.style.SUCCESS`, `self.style.WARNING`, and `self.style.ERROR` for colored terminal output."

In my project: `backend/management/commands/seed_demo.py`

```
# Line 124: Green success message
self.stdout.write(
    self.style.SUCCESS(f'  ✓ Created: {memory.title}')
)

# Line 139: Yellow warning message
self.stdout.write(
    self.style.WARNING('Note: Run Celery worker to analyze memories')
)
```

Why I did this: "Colors make terminal output scannable. Green means success, yellow means attention needed. This is standard Django practice."

---

2. Base Config

Interviewer: "Where is your Django project configured?" **You:** "In `settings.py`. I configured the database, installed apps, middleware stack, and REST Framework settings here."

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

```
# Line 6: I import standard library modules
import os
from pathlib import Path
from datetime import timedelta

# Line 9: I load environment variables
from dotenv import load_dotenv
load_dotenv()

# Line 15: BASE_DIR points to the 'backend' folder
BASE_DIR = Path(__file__).resolve().parent.parent
```

Why I did this: "All paths in Django are relative to `BASE_DIR`. Using `Path` instead of string concatenation is cleaner and cross-platform."

Sub-topic: SECRET_KEY

Interviewer: "How do you handle the Django secret key?" **You:** "I load it from environment variables. I never commit the real key to Git."

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

```
# Line 18: I get SECRET_KEY from environment, with a fallback for development
SECRET_KEY = os.getenv('SECRET_KEY', 'django-insecure-dev-key-change-in-production')
```

Why I did this: "The SECRET_KEY is used for cryptographic signing. If it leaks, attackers could forge session cookies. The fallback is clearly marked 'insecure' so I remember to set a real one in production."

Sub-topic: DEBUG

Interviewer: "How do you toggle debug mode?" **You:** "I read it from environment and convert the string to a boolean."

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

```
# Line 19: I compare the string to 'True' to get a boolean
DEBUG = os.getenv('DEBUG', 'True') == 'True'

# Line 22-25: I adjust ALLOWED_HOSTS based on DEBUG
if DEBUG:
    ALLOWED_HOSTS = ['*']
else:
    ALLOWED_HOSTS = os.getenv('ALLOWED_HOSTS', 'localhost,127.0.0.1').split(',')
```

Why I did this: "Environment variables are always strings. `os.getenv('DEBUG')` returns `'True'` not `True`. The `== 'True'` comparison converts it to a proper boolean."

Sub-topic: INSTALLED_APPS

Interviewer: "How do you register apps in Django?" **You:** "I add them to the `INSTALLED_APPS` list. Order matters for some apps."

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

```
# Line 34: All apps Django should load
INSTALLED_APPS = [
    # Line 36: Daphne MUST be first for async/websocket support
    'daphne',

    # Line 37-42: Django's built-in apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
```

```
    'django.contrib.staticfiles',

    # Line 44-47: Third-party packages
    'rest_framework',
    'corsheaders',
    'django_filters',
    'channels',

    # Line 50-54: My custom apps using AppConfig format
    'users.apps.UsersConfig',
    'memories.apps.MemoriesConfig',
    'notifications.apps.NotificationsConfig',
    'nlp.apps.NlpConfig',
]
```

Why I did this: "Using `UsersConfig` instead of just `'users'` lets me define custom app behavior in `apps.py`, like auto-loading signals."

---

3. Static Files

Interviewer: "How do you serve CSS/JS files in Django?" **You:** "I configure `STATIC_URL` and `STATIC_ROOT`. In production, I use **WhiteNoise** middleware."

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

```
# Line 121: The URL prefix for static files
STATIC_URL = '/static/'

# Line 122: Where collectstatic puts files for production
STATIC_ROOT = BASE_DIR / 'staticfiles'
```

Why I did this: "`STATIC_URL` is what appears in HTML (`/static/css/style.css`). `STATIC_ROOT` is the actual folder on disk where `collectstatic` copies files."

Sub-topic: WhiteNoise

Interviewer: "How do you serve static files in production?" **You:** "I use WhiteNoise middleware. It serves files directly from Django without needing Nginx."

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

```
# Line 59: WhiteNoise must be after SecurityMiddleware
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',  # <- Here
    # ... other middleware ...
]
```

Why I did this: "WhiteNoise compresses and caches static files. It's simpler than configuring Nginx for static serving and works on platforms like Heroku."

---

4. Media Files

Interviewer: "How do you handle user-uploaded files?" **You:** "I configure `MEDIA_URL` and `MEDIA_ROOT`. Files go to a separate folder from static files."

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

```
# Line 123: URL prefix for user uploads
MEDIA_URL = '/media/'

# Line 124: Actual folder on disk
MEDIA_ROOT = BASE_DIR / 'media'
```

Why I did this: "Static files are part of my code (CSS, JS). Media files are user uploads (avatars, journal images). They need separate handling because media is dynamic."

Sub-topic: upload_to

Interviewer: "How do you organize uploaded files into folders?" **You:** "I use the `upload_to` parameter with date formatting."

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

```
# Line 46: I use strftime-style formatting for dynamic paths
media_file = models.FileField(
    upload_to='memories/%Y/%m/%d/',  # Creates: media/memories/2024/12/14/
    blank=True,
    null=True
)
```

Why I did this: "Organizing by date prevents any single folder from having millions of files. It also makes it easy to find uploads from a specific day."

---

5. Templates

Interviewer: "Did you use Django Templates?" **You:** "Yes, I configured the template engine. It's mainly used for the Admin panel."

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

```
# Line 72: Template engine configuration
TEMPLATES = [
    {
        # Line 74: I use Django's built-in template engine
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
```

```
        # Line 75: Additional folders to search for templates
        'DIRS': [BASE_DIR / 'templates'],

        # Line 76: Automatically find templates in app folders
        'APP_DIRS': True,

        # Line 77-84: Context processors add variables to all templates
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Why I did this: " `APP_DIRS: True` means Django looks for `templates/` folder inside each app. Context processors automatically inject `request`, `user`, and `messages` into every template."

---

6. Middleware

Interviewer: "Explain Middleware with an example from your code." **You:** "Middleware is code that runs **before** and **after** every single request. I wrote a custom middleware to measure performance."

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

```
# Line 5: I import time for measuring duration
import time
import logging

# Line 8: I create a logger for this module
logger = logging.getLogger(__name__)

# Line 11: Middleware class definition
class RequestTimingMiddleware:
    # Line 22: __init__ is called once when Django starts
    def __init__(self, get_response):
        # Line 23: I store the next middleware/view in the chain
        self.get_response = get_response

    # Line 26: __call__ is called for EVERY request
    def __call__(self, request):
        # Line 29: BEFORE the view - I start the timer
        start_time = time.time()

        # Line 31: I pass the request to the view (or next middleware)
        response = self.get_response(request)
```

```
        # Line 35: AFTER the view - I calculate duration
        duration = time.time() - start_time

        # Line 36: I add a custom header to the response
        response['X-Process-Time'] = f"{duration:.4f}s"

        # Line 39-40: I log slow requests for debugging
        if duration > 1.0:
            logger.warning(f"Slow request: {request.path} took {duration:.4f}s")

        # Line 42: I return the response
        return response
```

Why I did this: "This allows me to monitor the latency of every API call in production without adding logging code to every single view function."

Sub-topic: init

Interviewer: "What happens in the middleware's `__init__` ?" **You:** "It's called once when Django starts. I store the `get_response` callable which represents the next step in the chain."

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

```
# Line 22: Called once at server startup
def __init__(self, get_response):
    # Line 23: get_response could be the next middleware OR the final view
    self.get_response = get_response
    # I could also do one-time setup here (load configs, etc.)
```

Why I did this: "Django passes `get_response` so my middleware can call the next step. Without storing it, I couldn't forward the request."

Sub-topic: call

Interviewer: "What is the `__call__` method?" **You:** "It makes the class callable like a function. Django calls it for every HTTP request."

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

```
# Line 26: Makes this class callable: middleware(request)
def __call__(self, request):
    # BEFORE view logic here
    start_time = time.time()

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

    # AFTER view logic here
    duration = time.time() - start_time
```

```
    response['X-Process-Time'] = f"{duration:.4f}s"

    return response
```

Why I did this: "This pattern lets me run code before AND after the view. I start the timer before, stop it after, then modify the response."

---

7. Authentication

Interviewer: "How did you handle user authentication?" **You:** "I created a **Custom User Model** extending `AbstractUser`."

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

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

# Line 9: My custom User model
class User(AbstractUser):
    # Line 15: I add fields that don't exist in the default model
    bio = models.TextField(blank=True, null=True, help_text="User biography")

    # Line 16: Avatar image with custom upload path
    avatar = models.ImageField(upload_to='avatars/', blank=True, null=True)

    # Line 17-18: Timestamps
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    # Line 20-23: Metadata
    class Meta:
        db_table = 'auth_user'  # Keep the standard table name
        verbose_name = 'User'
        verbose_name_plural = 'Users'
```

Why I did this: "If I used the default User model, adding 'Bio' or 'Avatar' later would require ugly one-to-one relationships. Extending `AbstractUser` gives me a clean, single table."

Sub-topic: AUTH_USER_MODEL

Interviewer: "How does Django know to use your custom User?" **You:** "I set `AUTH_USER_MODEL` in settings."

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

```
# Line 105: Tell Django to use my custom User model
AUTH_USER_MODEL = 'users.User'
```

Why I did this: "This setting MUST be done before the first migration. It tells Django's auth system, admin, and all contrib apps to use my model instead of the default."

Sub-topic: AUTH_PASSWORD_VALIDATORS

Interviewer: "How do you enforce password security?" **You:** "I use Django's built-in password validators."

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

```
# Line 107-112: Password validation rules
AUTH_PASSWORD_VALIDATORS = [
    # Prevents password too similar to username/email
    {'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'},

    # Minimum 8 characters by default
    {'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator'},

    # Rejects common passwords like '123456'
    {'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator'},

    # Rejects all-numeric passwords
    {'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator'},
]
```

Why I did this: "These validators run during user creation and password changes. They prevent weak passwords without me writing validation logic."

---

8. Caching

Interviewer: "Did you implement caching?" **You:** "I use Django's default in-memory cache. For production, I would configure Redis."

In my project: `backend/config/settings.py` *(Default behavior)*

```
# Django uses LocMemCache by default if CACHES is not defined
# For production, I would add:

# CACHES = {
#     'default': {
#         'BACKEND': 'django.core.cache.backends.redis.RedisCache',
#         'LOCATION': os.getenv('REDIS_URL', 'redis://localhost:6379/1'),
#     }
# }
```

Why I did this: "LocMemCache is fine for development. In production, Redis provides persistent, shared caching across multiple server instances."

---

9. Routing

Interviewer: "How do you organize your URLs?" **You:** "I use a central `urls.py` that delegates to app-specific URL files."

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

```
# Line 5-6: I import path and include
from django.urls import path, include

# Line 10: Main URL patterns
urlpatterns = [
    # Line 11: Admin site at /admin/
    path('admin/', admin.site.urls),

    # Line 12: Auth endpoints delegate to users app
    path('api/v1/auth/', include('users.urls')),

    # Line 13: Memory endpoints delegate to memories app
    path('api/v1/memories/', include('memories.urls')),
]
```

Why I did this: "This keeps the main `urls.py` clean. Each app manages its own routes. The `api/v1/` prefix provides URL versioning."

Sub-topic: path()

Interviewer: "What does `path()` do?" **You:** "It maps a URL pattern to a view or another URL configuration."

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

```
# Line 20: path takes: (URL pattern, view, optional name)
path('register/', register_view, name='register'),

# The name='register' lets me use reverse('register') in code
```

Why I did this: "Named URLs are essential. I can change the URL pattern later without breaking code that uses `reverse('register')`."

Sub-topic: include()

Interviewer: "What is `include()` for?" **You:** "It delegates URL handling to another URLconf (another `urls.py` file)."

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

```
# Line 12: Everything starting with 'api/v1/auth/' goes to users.urls
path('api/v1/auth/', include('users.urls')),

# So 'api/v1/auth/login/' is handled by users/urls.py
```

Why I did this: "Modularity. The `users` app doesn't know or care that it's mounted at `/api/v1/auth/`. It just defines `/login/`, `/register/`, etc."

---

10. URL Params

Interviewer: "How do you capture URL parameters?" **You:** "DRF Routers handle this automatically. For custom URLs, I use path converters or query params."

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

```
# Line 77: I get query parameters from request.query_params
query = request.query_params.get('q', '')

# The URL is: /api/v1/memories/search/?q=happy
# request.query_params.get('q') returns 'happy'
```

Why I did this: "Query params are for filtering/searching. Path params (`/memories/123/`) are for identifying specific resources."

---

11. Admin URL

Interviewer: "Did you customize the Django Admin?" **You:** "Yes, I registered my models with custom display options."

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

```
# Line 4: I use the decorator to register the model
@admin.register(Memory)
class MemoryAdmin(admin.ModelAdmin):
    # Line 6: Columns to show in the list view
    list_display = ('id', 'user', 'title', 'media_type', 'analysis_status', 'created_at')

    # Line 7: Sidebar filters
    list_filter = ('media_type', 'analysis_status', 'created_at')

    # Line 8: Search box searches these fields
    search_fields = ('title', 'text', 'user__username')

    # Line 9: Fields that can't be edited
    readonly_fields = ('processed_metrics_json', 'embedding_preview', 'created_at',
'updated_at')

    # Line 10: Default sort order
    ordering = ('-created_at',)
```

Why I did this: "Custom admin makes it easy to debug and manage data without building a separate admin UI. I can filter by status, search by title, and view read-only AI results."

---

12. Model Definition

Interviewer: "Explain your data models." **You:** "I have `User` and `Memory` models. Memory stores journal entries."

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

```
# Line 12: I define a Django model by inheriting from models.Model
class Memory(models.Model):
    # Line 18-24: I define choices as a tuple of tuples
    MEDIA_TYPE_CHOICES = (
        ('text', 'Text Only'),
        ('photo', 'Photo'),
        ('audio', 'Audio'),
        ('video', 'Video'),
    )

    # Line 35: ForeignKey links Memory to User
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='memories')

    # Line 36: Standard text fields
    title = models.CharField(max_length=255)
    text = models.TextField()
```

Why I did this: "Each class represents a database table. Django's ORM converts Python objects to SQL queries automatically."

---

13. Fields

Interviewer: "What types of fields did you use?" **You:** "I used a variety: `CharField`, `TextField`, `JSONField`, `FileField`, `DateTimeField`."

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

```
# Line 36: CharField for short text (max 255 chars)
title = models.CharField(max_length=255, help_text="Memory title")

# Line 37: TextField for unlimited text
text = models.TextField(help_text="Journal entry text")

# Line 54-58: JSONField for flexible structured data
metrics_json = models.JSONField(
    default=dict,      # Empty dict {} if not set
    blank=True,        # OK to leave empty in forms
    help_text="AI analysis results"
)

# Line 72-73: DateTimeField with auto timestamps
created_at = models.DateTimeField(auto_now_add=True)  # Set once on create
updated_at = models.DateTimeField(auto_now=True)       # Updated on every save
```

Why I did this: "Each field type maps to a specific database column type. `auto_now_add` and `auto_now` eliminate manual timestamp handling."

---

14. Relationships

Interviewer: "How did you link Users to Memories?" **You:** "I used a `ForeignKey` with CASCADE delete behavior."

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

```
# Line 35: ForeignKey creates a many-to-one relationship
user = models.ForeignKey(
    User,                          # The related model
    on_delete=models.CASCADE,      # If user deleted, delete their memories
    related_name='memories'        # Allows: user.memories.all()
)
```

Why I did this: " `CASCADE` ensures no orphaned memories exist. `related_name` lets me access memories from user objects: `user.memories.filter(analysis_status='done')` ."

---

15. ORM

Interviewer: "Show me an ORM query from your project." **You:** "I use the ORM extensively. Here's a complex example with annotations."

In my project: `backend/nlp/search.py`

```
# Line 89: I import ORM tools for complex queries
from django.db.models import Q, Case, When, IntegerField

# Line 147-152: I annotate each memory with a calculated score
memories = Memory.objects.filter(user=user).annotate(
    # For each search term, add 1 if it matches
    match_score=sum(
        Case(When(Q(text__icontains=term), then=1), default=0, output_field=IntegerField())
        for term in terms
    )
)

# Line 155: I filter for score > 0 and sort by score descending
results = memories.filter(match_score__gt=0).order_by('-match_score')
```

Why I did this: "ORM queries are safer than raw SQL (no injection). `annotate` lets me add computed columns, and `Case/When` creates conditional logic in SQL."

---

16. File Uploads

Interviewer: "How do you handle file uploads?" **You:** "I use `FileField` with dynamic paths and validate formats."

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

```
# Line 46-51: FileField for any uploaded file
media_file = models.FileField(
    upload_to='memories/%Y/%m/%d/',  # Dynamic path with date
    blank=True,
    null=True,
    help_text="Attached media (image, audio, or video)"
)
```

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

```
# Line 174: I set max upload size (50MB)
MAX_UPLOAD_SIZE = 52428800

# Line 175-177: I define allowed formats
ALLOWED_AUDIO_FORMATS = ['mp3', 'wav', 'ogg', 'm4a']
ALLOWED_IMAGE_FORMATS = ['jpg', 'jpeg', 'png', 'gif', 'webp']
ALLOWED_VIDEO_FORMATS = ['mp4', 'avi', 'mov', 'mkv']
```

Why I did this: "Dynamic `upload_to` prevents folder bloat. Format validation prevents users from uploading executable files disguised as images."

---

17. FBV (Function Based Views)

Interviewer: "Did you use Function Based Views?" **You:** "Yes, for simple endpoints like health checks."

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

```
# Line 6: I import the api_view decorator
from rest_framework.decorators import action, api_view
from rest_framework.response import Response

# Line 20: @api_view defines allowed HTTP methods
@api_view(['GET'])
def system_health(request):
    # Line 26-30: Simple function that returns a Response
    return Response({
        'status': 'healthy',
        'system': 'VibeVault Backend',
        'version': '1.0.0'
    })
```

Why I did this: "For a simple endpoint with no model operations, a function is cleaner than a full class. `@api_view` adds DRF features like content negotiation."

DRF (Django Rest Framework) TOPICS

---

18. DRF Auth

Interviewer: "How does your API authenticate requests?" **You:** "I use **JWT Authentication** via SimpleJWT."

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

```
# Line 129-135: REST Framework global settings
REST_FRAMEWORK = {
    # Line 130-132: JWT as the default auth method
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    # Line 133-135: Require login by default
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}

# Line 146-151: JWT token settings
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=1),   # Short-lived access token
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),   # Longer-lived refresh token
    'ALGORITHM': 'HS256',                           # Signing algorithm
    'SIGNING_KEY': SECRET_KEY,                      # Use Django's secret key
}
```

Why I did this: "JWT is stateless - the server doesn't store sessions. The client includes the token in every request's `Authorization: Bearer <token>` header."

---

19. DRF Pagination

Interviewer: "How do you handle large datasets?" **You:** "I configured global PageNumberPagination."

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

```
# Line 141-142: Pagination settings
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 20,  # 20 items per page
}
```

Response format:

```
{
    "count": 150,
    "next": "/api/v1/memories/?page=2",
    "previous": null,
```

```
    "results": [...]
}
```

Why I did this: "Without pagination, loading 10,000 memories would crash the browser. The client gets 20 at a time and can request more."

---

20. DRF Throttling

Interviewer: "Did you implement rate limiting?" **You:** "Not yet, but I know how. I would add throttle classes in production."

In my project: *(Would add to `settings.py`)*

```
# Production throttling config:
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',  # For non-logged-in users
        'rest_framework.throttling.UserRateThrottle'   # For logged-in users
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',   # 100 requests per day for anonymous
        'user': '1000/day'   # 1000 requests per day for authenticated
    }
}
```

Why: "Throttling prevents abuse. Without it, someone could make millions of requests and overload the server."

---

21. DRF Versioning

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

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

```
# Line 12-13: All routes prefixed with api/v1/
urlpatterns = [
    path('api/v1/auth/', include('users.urls')),
    path('api/v1/memories/', include('memories.urls')),
]

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

Why I did this: "Versioning lets me make breaking changes without breaking existing clients. Old clients use `/v1/`, new clients use `/v2/`."

---

22. CORS

Interviewer: "How does your React frontend talk to Django?" **You:** "I use `django-cors-headers` to allow cross-origin requests."

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

```
# Line 46: Install the corsheaders app
INSTALLED_APPS = [
    'corsheaders',
    # ...
]

# Line 61: CorsMiddleware must be high in the stack
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    # ...
]

# Line 28-31: Define allowed origins
CORS_ALLOWED_ORIGINS = os.getenv(
    'CORS_ALLOWED_ORIGINS',
    'http://localhost:5173,http://localhost:3000'  # Vite and CRA dev servers
).split(',')
```

Why I did this: "By default, browsers block requests from `localhost:5173` (React) to `localhost:8000` (Django). CORS headers tell the browser it's okay."

---

23. DRF Routers

Interviewer: "How do you generate URLs for ViewSets?" **You:** "I use DefaultRouter which creates CRUD endpoints automatically."

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

```
# Line 6: I import DefaultRouter
from rest_framework.routers import DefaultRouter

# Line 9: I create a router instance
router = DefaultRouter()

# Line 10: I register my ViewSet
router.register(r'', MemoryViewSet, basename='memory')

# Line 12-13: Include router URLs in urlpatterns
urlpatterns = [
    path('', include(router.urls)),
]

# This automatically creates:
# GET    /memories/         -> list()
# POST   /memories/         -> create()
```

```
# GET    /memories/{id}/    -> retrieve()
# PUT    /memories/{id}/    -> update()
# PATCH  /memories/{id}/    -> partial_update()
# DELETE /memories/{id}/    -> destroy()
```

Why I did this: "Routers enforce REST conventions. I get 6 endpoints with 2 lines of code instead of writing 6 separate `path()` calls."

---

24. ViewSet Mapping

Interviewer: "Can you use ViewSets without a Router?" **You:** "Yes, I map actions to URLs manually using `.as_view()`."

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

```
# Line 15-17: I map HTTP methods to ViewSet actions
register_view = RegisterView.as_view({'post': 'register'})
profile_me = ProfileView.as_view({'get': 'me'})
profile_update = ProfileView.as_view({'patch': 'update_profile'})

# Line 19-25: Then use them in urlpatterns
urlpatterns = [
    path('register/', register_view, name='register'),
    path('profile/', profile_me, name='profile'),
    path('profile/update/', profile_update, name='profile_update'),
]
```

Why I did this: "For non-standard CRUD operations (like `register`), manual mapping gives more control. The dict maps HTTP methods to action names."

---

25. Serializer Mapping

Interviewer: "How do you convert model data to JSON?" **You:** "I use ModelSerializer which generates fields from the model."

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

```
# Line 9: I inherit from ModelSerializer
class MemorySerializer(serializers.ModelSerializer):
    # Line 11: I customize how the user field appears
    user = serializers.StringRelatedField(read_only=True)  # Shows "username (email)"

    # Line 15-22: Meta defines the model and fields
    class Meta:
        model = Memory
        fields = (
            'id', 'user', 'title', 'text', 'media_type', 'media_file',
            'metrics_json', 'embedding', 'analysis_status',
            'emotion', 'sentiment', 'created_at', 'updated_at'
```

```
        )
        # Line 22: These fields can't be modified via API
        read_only_fields = ('id', 'user', 'metrics_json', 'embedding', 'analysis_status',
'created_at', 'updated_at')
```

Why I did this: "ModelSerializer inspects the model and creates appropriate field types automatically. `read_only_fields` prevents clients from tampering with system-generated data."

---

26. Nested Serialization

Interviewer: "How do you display related or computed data?" **You:** "I use SerializerMethodField to extract and transform data."

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

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

    # Line 24-26: The method name MUST be get_<fieldname>
    def get_emotion(self, obj):
        # obj is the Memory instance being serialized
        return obj.emotion  # This calls the @property on the model

    def get_sentiment(self, obj):
        return obj.sentiment
```

In my project: `backend/memories/models.py` (The property being called)

```
# Line 87-92: Model property that extracts from JSON
@property
def emotion(self):
    if self.metrics_json and 'emotion' in self.metrics_json:
        return self.metrics_json['emotion'].get('top_emotion', 'neutral')
    return None
```

Why I did this: "The raw JSON has nested structure (`metrics_json.emotion.top_emotion`). SerializerMethodField lets me flatten it to just `emotion: 'joy'` for the frontend."

✅ Complete Summary

All **26 topics** covered with:

- Main concept explanation
- Sub-topics with code
- Line-by-line comments
- "Why I did this" rationale