Scopio Project - Complete Explanation

A Beginner-Friendly Guide to Understanding Everything

Table of Contents

---

1. Project Overview

What Problem Are We Solving?

Scopio is a **learning platform** where users can:

- Sign up and log in (with email/password OR Google)
- Access courses and videos
- Read articles
- Track their progress

Why This Architecture?

Backend (Django): Handles database, authentication, business logic **Frontend (React)**: Provides the user interface **PostgreSQL (Neon)**: Stores all data persistently

This is called a **"decoupled" or "API-based" architecture** - the frontend and backend are separate applications that communicate via HTTP requests.

---

2. Project Structure

```
Scopio/
├── Backend/                 # Django backend application
│   ├── main/               # Main Django settings & configuration
│   │   ├── settings.py     # ALL configuration lives here
│   │   ├── urls.py         # Main URL routing
```

```
│   │   └── wsgi.py          # Production server entry point
│   │
│   ├── api/                 # User authentication API
│   │   ├── models.py        # Database table definitions (uses Django's User)
│   │   ├── serializers.py   # Converts data ↔ JSON
│   │   ├── views.py         # Handles HTTP requests
│   │   └── urls.py          # API endpoint routes
│   │
│   ├── glogin/              # Google OAuth integration
│   │   ├── views.py         # OAuth flow handlers
│   │   ├── adapter.py       # Links Google accounts to users
│   │   └── urls.py          # OAuth routes
│   │
│   ├── video/               # Video/course management
│   │   ├── models.py        # Video & Demo database models
│   │   ├── serializers.py   # Video data → JSON conversion
│   │   ├── views.py         # Video API endpoints
│   │   └── urls.py          # Video routes
│   │
│   ├── db.sqlite3           # Local development database (NOT USED)
│   ├── manage.py            # Django command-line tool
│   └── requirements.txt     # Python dependencies
│
├── frontend/                # React frontend application
│   ├── src/
│   │   ├── components/      # Reusable UI components
│   │   │   ├── Login.jsx    # Login form
│   │   │   ├── Signup.jsx   # Registration form
│   │   │   ├── Navbar.jsx   # Navigation bar
│   │   │   └── ...
│   │   ├── pages/           # Full page components
│   │   ├── api.js           # Backend communication setup
│   │   ├── constants.js     # App-wide constants
│   │   └── App.jsx          # Main application component
│   │
│   ├── package.json         # JavaScript dependencies
│   └── vite.config.js       # Build tool configuration
│
├── benv/                    # Python virtual environment
└── docker-compose.yaml      # Container orchestration
```

---

3. Backend Architecture

3.1 What is Django?

Django is a **Python web framework** that helps you build web applications quickly. Think of it as a toolkit with pre-built components for common tasks (authentication, database, admin panel, etc.).

3.2 Django Apps

Your Django project is divided into **"apps"** - each app handles one area of functionality:

1. `main`: Core settings and configuration
2. `api`: User registration, login, and JWT tokens
3. `glogin`: Google OAuth authentication
4. `video`: Video/course management

3.3 Django REST Framework (DRF)

We use **Django REST Framework** to build APIs. It provides:

- **Serializers**: Convert complex data to JSON and vice versa
- **ViewSets/Views**: Handle HTTP requests (GET, POST, PUT, DELETE)
- **Authentication**: JWT token validation
- **Permissions**: Control who can access what

---

4. Database Setup

4.1 What Database Are We Using?

PostgreSQL on Neon Cloud (hosted database)

Connection string in `settings.py`:

```
DATABASES = {
    'default': dj_database_url.parse(
        "postgresql://neondb_owner:npg_1PFVp6gXeQjy@..."
    )
}
```

4.2 Database Tables (Models)

Django uses **Models** to define database tables. Each model = one table.

User Model (Built-in Django)

```
Table: auth_user
Columns:
- id (primary key)
- username (unique)
- email (unique)
- password (hashed)
- first_name
- last_name
- date_joined
- is_active
- is_staff
- is_superuser
```

Video Model (Custom)

```
# video/models.py
class Video(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    video_url = models.URLField()
    thumbnail = models.URLField()
    created_at = models.DateTimeField(auto_now_add=True)
```

Becomes:

```
Table: video_video
Columns:
- id
- title
- description
- video_url
- thumbnail
- created_at
```

SocialAccount Model (django-allauth)

```
Table: socialaccount_socialaccount
Columns:
- id
- user_id (foreign key → auth_user)
- provider (e.g., "google")
- uid (Google user ID)
- extra_data (JSON with Google profile info)
```

4.3 Migrations

Migrations are Django's way of updating the database structure.

```
python manage.py makemigrations  # Creates migration files
python manage.py migrate         # Applies changes to database
```

When you run migrate:

1. Django reads migration files (`0001_initial.py`, etc.)
2. Converts them to SQL (`CREATE TABLE`, `ALTER TABLE`, etc.)
3. Executes SQL on the database

5. Authentication System Deep Dive

5.1 Two Types of Authentication

1. **Form-Based**: Email/Username + Password
2. **OAuth-Based**: "Continue with Google"

5.2 JWT Tokens (JSON Web Tokens)

What is a JWT?

A JWT is a **secure, self-contained token** that proves who you are.

Structure:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJleHAiOjE2ODk5ODc2MDB9.xyz1
23...
  ↑                                    ↑                                    ↑
  Header (algorithm)                   Payload (user data)
Signature (verification)
```

Parts:

1. **Header**: Token type and algorithm
2. **Payload**: User data (user_id, expiration time)
3. **Signature**: Cryptographic signature to prevent tampering

Token Types in Our App

1. **Access Token**:
   - Short-lived (30 minutes)
   - Used for API requests
   - Sent in `Authorization: Bearer <token>` header
2. **Refresh Token**:
   - Long-lived (1 day)
   - Used to get new access tokens
   - Stored securely (HttpOnly cookie or localStorage)

Settings Configuration

```
# settings.py
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(minutes=30),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=1),
    "ROTATE_REFRESH_TOKENS": True,      # Generate new refresh token on each
refresh
```

```
    "BLACKLIST_AFTER_ROTATION": True,   # Old refresh tokens can't be reused
}
```

5.3 Session vs Token Authentication

Sessions (Traditional):

- Server stores login state in database
- Client gets session ID cookie
- Server checks session on each request
- **Used for**: Django admin panel

JWT Tokens (Modern):

- Server doesn't store state
- Client stores token (localStorage/cookies)
- Token contains all needed info
- **Used for**: API authentication

Our app uses **BOTH**:

- Sessions for Django admin (`/admin/`)
- JWT tokens for API endpoints (`/api/`)

---

6. Django Components Explained

6.1 What is a Serializer?

Serializers are translators between Python objects and JSON.

Example: UserSerializer

```
# api/serializers.py
from django.contrib.auth.models import User
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'first_name', 'last_name',
'password']
        extra_kwargs = {'password': {'write_only': True}}
```

What this does:

Python → JSON (Serialization):

```
user = User.objects.get(id=1)
serializer = UserSerializer(user)
print(serializer.data)
```

Output:

```
{
  "id": 1,
  "username": "john",
  "email": "john@example.com",
  "first_name": "John",
  "last_name": "Doe"
}
```

JSON → Python (Deserialization):

```
data = {
    "username": "jane",
    "email": "jane@example.com",
    "password": "secret123"
}
serializer = UserSerializer(data=data)
if serializer.is_valid():
    user = serializer.save()  # Creates User object
```

Validation in Serializers

```
def validate_email(self, value):
    """Custom validation for email field"""
    if User.objects.filter(email__iexact=value).exists():
        raise serializers.ValidationError(
            "This email is already registered."
        )
    return value
```

When validation runs:

- When you call `serializer.is_valid()`
- Before saving data

Why we validate:

- Prevent duplicate emails/usernames
- Check if user already has Google account
- Ensure password meets requirements

6.2 What is a View?

Views handle HTTP requests and return responses.

Types of Views

1. Function-Based Views (FBV):

```
@api_view(['GET'])
@permission_classes([AllowAny])
def get_csrf_token(request):
    return JsonResponse({"detail": "CSRF cookie set"})
```

2. Class-Based Views (CBV):

```
class CreateUserView(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [AllowAny]
```

Request-Response Cycle

```
1. Browser sends:
   POST /api/user/register/
   Body: {"username": "john", "email": "john@example.com", ...}

2. Django routes to: CreateUserView

3. View process:
   - Get data from request.data
   - Create serializer with data
   - Validate data (serializer.is_valid())
   - Save to database (serializer.save())
   - Return response

4. Browser receives:
   Status: 201 Created
   Body: {"message": "User created successfully", "user": {...}}
```

6.3 What are URL Patterns?

URLs connect **HTTP paths** to **views**.

```
# api/urls.py
urlpatterns = [
```

```
    path('auth/login/', CookieTokenObtainPairView.as_view(),
name='cookie_token_obtain_pair'),
    path('auth/csrf/', get_csrf_token, name='csrf_token'),
    path('users/', CreateUserView.as_view()),
]
```

URL matching:

```
Request: POST http://localhost:8000/api/auth/login/
         ↓
Matches: path('auth/login/', ...)
         ↓
Calls:   CookieTokenObtainPairView.post()
```

6.4 What are Permissions?

Permissions control who can access what.

```
permission_classes = [AllowAny]      # Anyone, even not logged in
permission_classes = [IsAuthenticated]  # Must have valid JWT token
```

How it works:

```
# In view
class VideoListView(generics.ListAPIView):
    permission_classes = [IsAuthenticated]  # Require login

# When request comes:
1. DRF checks Authentication header
2. Extracts JWT token
3. Validates token (signature, expiration)
4. If valid → Allow access
5. If invalid → Return 401 Unauthorized
```

6.5 What are Middleware?

Middleware are functions that process **every request** before it reaches your view.

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',    # Handle CORS
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',  # Session handling
    'django.middleware.csrf.CsrfViewMiddleware',  # CSRF protection
    'django.contrib.auth.middleware.AuthenticationMiddleware',  # User auth
```

```
    'allauth.account.middleware.AccountMiddleware',  # OAuth handling
]
```

Request flow through middleware:

```
Browser Request
   ↓
CorsMiddleware (checks if frontend is allowed)
   ↓
SessionMiddleware (loads session data)
   ↓
CsrfViewMiddleware (validates CSRF token)
   ↓
AuthenticationMiddleware (sets request.user)
   ↓
Your View
   ↓
Response back through middleware
   ↓
Browser Response
```

---

7. API Endpoints

7.1 User Registration

Endpoint: `POST /api/user/register/`

Request:

```
{
  "username": "john",
  "email": "john@example.com",
  "first_name": "John",
  "last_name": "Doe",
  "password": "secure123"
}
```

What happens internally:

1. **Request arrives** → `CreateUserView`
2. **Validation** → `UserSerializer.validate_email()`
   - Check if email exists
   - Check if email has Google account
3. **If valid** → `serializer.save()`
   - Calls `User.objects.create_user()`
   - Hashes password (Django security)

- Saves to database

4. **Response**:

```
{
  "message": "User created successfully",
  "user": {
    "id": 1,
    "username": "john",
    "email": "john@example.com"
  }
}
```

7.2 Login (Form-based)

Endpoint: `POST /api/auth/login/`

Request:

```
{
  "username": "john",  // or email
  "password": "secure123"
}
```

Internal flow:

1. **CookieTokenObtainPairView.post()**
2. **Check if user exists**:

```
user = User.objects.filter(
    Q(username=username) | Q(email=username)
).first()
```

3. **Check if OAuth-only account**:

```
if user and not user.has_usable_password():
    # Has Google account, no password
    return Error("Use Google to login")
```

4. **Validate credentials**:
   - Django checks password hash
   - If match → Generate JWT tokens
5. **Generate tokens**:

```
refresh = RefreshToken.for_user(user)
access = str(refresh.access_token)
```

6. **Set cookies**:

```
response.set_cookie("access", access, httponly=True)
response.set_cookie("refresh", refresh, httponly=True)
```

7. **Response**: `{"detail": "login successful"}`

7.3 CSRF Token Endpoint

Endpoint: `GET /api/auth/csrf/`

Why needed: Cross-Site Request Forgery protection

What it does:

```
@ensure_csrf_cookie
def get_csrf_token(request):
    return JsonResponse({"detail": "CSRF cookie set"})
```

This endpoint just sets a cookie: `csrftoken=abc123...`

Frontend usage:

```
// On app load
await api.get('/api/auth/csrf/');  // Gets CSRF cookie

// On form submit
// Axios automatically reads cookie and adds header:
// X-CSRFToken: abc123...
```

---

8. Frontend-Backend Connection

8.1 Axios Configuration (api.js)

```
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL,  // http://localhost:8000
  withCredentials: true,  // Send cookies with requests
});
```

What `withCredentials: true` does:

- Sends cookies (session, CSRF) with every request
- Allows backend to read/set cookies

8.2 Request Interceptors

Interceptors modify requests/responses automatically.

```
// REQUEST INTERCEPTOR
api.interceptors.request.use((config) => {
  // 1. Add JWT token to header
  const token = localStorage.getItem('access');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }

  // 2. Add CSRF token for POST/PUT/DELETE
  const unsafeMethods = ['POST', 'PUT', 'PATCH', 'DELETE'];
  if (unsafeMethods.includes(config.method?.toUpperCase())) {
    const csrfToken = getCsrfToken();  // Read from cookie
    if (csrfToken) {
      config.headers['X-CSRFToken'] = csrfToken;
    }
  }

  return config;
});
```

What this means: Every API call automatically gets:

- JWT token in Authorization header
- CSRF token in X-CSRFToken header

8.3 CORS (Cross-Origin Resource Sharing)

The Problem:

- Frontend: `http://localhost:5173`
- Backend: `http://localhost:8000`
- Different ports = **different origins**
- Browsers block this by default (security)

The Solution (in Django):

```
# settings.py
CORS_ALLOWED_ORIGINS = ['http://localhost:5173']
CORS_ALLOW_CREDENTIALS = True  # Allow cookies
```

How it works:

```
1. Browser sends: OPTIONS /api/users/ (preflight request)
   Origin: http://localhost:5173

2. Backend responds:
   Access-Control-Allow-Origin: http://localhost:5173
   Access-Control-Allow-Credentials: true
   Access-Control-Allow-Methods: GET, POST, PUT, DELETE

3. Browser sees "OK" → Allows the actual request

4. Browser sends: GET /api/users/
   Origin: http://localhost:5173

5. Backend responds with data
```

8.4 Environment Variables

Frontend (`.env`):

```
VITE_API_URL=http://localhost:8000
```

Backend (`.env` or environment):

```
DJANGO_SECRET_KEY=your-secret-key
DEBUG=True
FRONTEND_URL=http://localhost:5173
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-secret
```

Why: Keep sensitive data out of code, easy to change per environment.

---

9. Complete Login Flows

9.1 Form-Based Login (Step-by-Step)

User types email + password, clicks "Login"

Frontend Actions:

1. **Component mounts** → Fetch CSRF token:

```
useEffect(() => {
  fetchCsrfToken();  // GET /api/auth/csrf/
}, []);
```

2. **User submits form**:

```
const handleSubmit = async (e) => {
  e.preventDefault();
  await login(formData.emailOrUsername, formData.password);
};
```

3. **login() function** (in api.js):

```
export async function login(username, password) {
  const { data } = await api.post('/api/auth/login/', {
    username,
    password,
  });
  return data;
}
```

4. **Axios interceptor adds**:
   - Header: `X-CSRFToken: abc123...`
   - Cookie: `csrftoken=abc123...`

Backend Actions:

5. **Django routing**:

```
POST /api/auth/login/ → CookieTokenObtainPairView
```

6. **View processing**:

```
def post(self, request):
    # Get credentials
    username = request.data.get('username')
    password = request.data.get('password')

    # Check if OAuth-only account
    user = User.objects.filter(
        Q(username=username) | Q(email=username)
    ).first()
```

```
    if user and not user.has_usable_password():
        return Response({
            "detail": "This account was created with Google..."
        }, status=400)

    # Validate credentials (parent class)
    response = super().post(request)

    # Generate JWT tokens
    refresh = RefreshToken.for_user(user)
    access = str(refresh.access_token)

    # Set cookies
    response.set_cookie("access", access, httponly=True, samesite="Lax")
    response.set_cookie("refresh", refresh, httponly=True, samesite="Lax")

    return response
```

7. **Database queries** (by Django):

```
SELECT * FROM auth_user
WHERE username = 'john' OR email = 'john';

-- Check password hash
-- If valid, get user data

SELECT * FROM socialaccount_socialaccount
WHERE user_id = 1 AND provider = 'google';
```

8. **Response sent**:

```
HTTP/1.1 200 OK
Set-Cookie: access=eyJhbGci...; HttpOnly; SameSite=Lax
Set-Cookie: refresh=eyJhbGci...; HttpOnly; SameSite=Lax
Content-Type: application/json

{"detail": "login successful"}
```

Frontend Receives Response:

9. **Success handler**:

```
try {
  await login(...);
  console.log('Login successful');
  onLoginSuccess();  // Navigate to home
} catch (error) {
```

```
    setErrors({
      general: error.response?.data?.detail
    });
  }
```

10. **Future API calls**:
    - Cookies automatically sent
    - JWT token in Authorization header
    - User is authenticated!

9.2 Google OAuth Login (Step-by-Step)

User clicks "Continue with Google"

Initial Flow:

1. **Frontend**:

```
const handleSocialLogin = (provider) => {
  if (provider === 'Google') {
    const backendURL = import.meta.env.VITE_API_URL;
    window.location.href = `${backendURL}/glogin/google/start/`;
  }
};
```

2. **Browser redirects** to:

```
http://localhost:8000/glogin/google/start/
```

3. **Backend** (`glogin/views.py`):

```
def google_start(request):
    next_url = '/glogin/google/finalize/'
    return redirect(f'/accounts/google/login/?process=login&next=
{next_url}')
```

4. **Django-allauth middleware**:
    - Reads Google credentials from settings:

```
SOCIALACCOUNT_PROVIDERS = {
    'google': {
        'APP': {
            'client_id': 'your-client-id',
```

```
            'secret': 'your-secret',
        }
    }
}
```

- Redirects to Google OAuth:

```
https://accounts.google.com/o/oauth2/v2/auth?
  client_id=your-client-id&
  redirect_uri=http://localhost:8000/accounts/google/login/callback/&
  response_type=code&
  scope=profile email
```

User at Google:

5. **Google login page shows**
6. **User signs in** / selects account
7. **Google asks**: "Allow Scopio to access your profile?"
8. **User clicks "Allow"**
9. **Google redirects back** with auth code:

```
http://localhost:8000/accounts/google/login/callback/?code=abc123xyz
```

Backend Processing:

10. **Django-allauth callback handler**:

```
# Automatic by allauth
# 1. Exchange code for access token
# 2. Get user profile from Google
# 3. Call adapter.pre_social_login()
```

11. **Custom adapter** (`glogin/adapter.py`):

```
def pre_social_login(self, request, sociallogin):
    # Get email from Google data
    email = sociallogin.user.email  # e.g., "john@gmail.com"

    # Check if user with this email exists
    try:
        existing_user = User.objects.get(email__iexact=email)
```

```
            # User exists! Link this Google account to them
            sociallogin.connect(request, existing_user)
            logger.info(f"Linked Google to existing user: {email}")
        except User.DoesNotExist:
            # New user, will be created automatically
            logger.info(f"New Google user: {email}")
```

12. **Database operations**:

```
-- Check if user exists
SELECT * FROM auth_user WHERE email = 'john@gmail.com';

-- If exists, link Google account
INSERT INTO socialaccount_socialaccount
(user_id, provider, uid, extra_data)
VALUES (1, 'google', '1234567890', '{"email": "john@gmail.com", ...}');

-- If new, create user
INSERT INTO auth_user (username, email, first_name, last_name)
VALUES ('john', 'john@gmail.com', 'John', 'Doe');
-- Then create social account link
```

13. **Allauth redirects** to `next` parameter:

```
/glogin/google/finalize/
```

14. **Finalize view** (`glogin/views.py`):

```
def google_finalize(request):
    if not request.user.is_authenticated:
        return redirect(f"{FRONTEND_URL}/?error=google_auth_failed")

    # Generate JWT tokens
    refresh = RefreshToken.for_user(request.user)
    access = str(refresh.access_token)
    refresh_str = str(refresh)

    # Redirect to frontend with tokens in URL hash
    return redirect(f"{FRONTEND_URL}/#access={access}&refresh=
{refresh_str}")
```

15. **Browser redirects** to:

```
http://localhost:5173/#access=eyJhbGci...&refresh=eyJhbGci...
```

Frontend Completes Flow:

16. **App.jsx useEffect**:

```
useEffect(() => {
  // Parse tokens from URL hash
  const hash = window.location.hash.substring(1);
  if (hash) {
    const params = new URLSearchParams(hash);
    const accessToken = params.get('access');
    const refreshToken = params.get('refresh');

    if (accessToken && refreshToken) {
      // Store tokens
      localStorage.setItem('access', accessToken);
      localStorage.setItem('refresh', refreshToken);

      // Navigate to Welcome page
      setShowWelcome(true);

      // Clean URL
      window.history.replaceState({}, '', '/');
    }
  }
}, []);
```

17. **User is now logged in!**
    - JWT tokens stored
    - All API requests authenticated
    - Can access protected routes

---

10. Security & Cookies

10.1 Cookie Attributes Explained

```
response.set_cookie(
    "access",
    access_token,
    httponly=True,      // JavaScript can't read it (XSS protection)
    secure=False,       // Only send over HTTPS (True in production)
    samesite="Lax",     // CSRF protection
    max_age=1800,       // Cookie expires in 30 minutes
)
```

HttpOnly:

- Prevents JavaScript from accessing cookie

- Protects against XSS attacks
- Cookie only sent in HTTP requests

Secure:

- Cookie only sent over HTTPS
- Not over HTTP (unencrypted)
- In development: `False` (localhost uses HTTP)
- In production: `True` (HTTPS required)

SameSite:

- `Lax`: Cookie sent in same-site requests (normal navigation)
- `None`: Cookie sent in cross-site requests (requires Secure=True)
- `Strict`: Cookie only sent from same domain

Our configuration:

```
# Development (localhost)
CSRF_COOKIE_SAMESITE = 'Lax'
CSRF_COOKIE_SECURE = False
SESSION_COOKIE_SAMESITE = 'Lax'
SESSION_COOKIE_SECURE = False

# Production (HTTPS)
CSRF_COOKIE_SAMESITE = 'None'  # Frontend different domain
CSRF_COOKIE_SECURE = True      # HTTPS only
SESSION_COOKIE_SAMESITE = 'None'
SESSION_COOKIE_SECURE = True
```

10.2 CSRF Protection

What is CSRF? Cross-Site Request Forgery - malicious site making requests as you.

Example attack (without protection):

```
<!-- Evil site -->
<form action="http://yourbank.com/transfer" method="POST">
  <input name="to" value="attacker-account">
  <input name="amount" value="1000">
</form>
<script>document.forms[0].submit();</script>
```

If you're logged into your bank, this would work!

Django's CSRF protection:

1. Backend generates random token
2. Sets cookie: `csrftoken=abc123`

3. Frontend reads cookie
4. Frontend sends header: `X-CSRFToken: abc123`
5. Backend validates: cookie matches header?
6. If yes → Allow request
7. If no → 403 Forbidden

Evil site can't read your cookies (browser security), so can't send valid CSRF token.

10.3 Password Hashing

Never store plain passwords!

```
# When user registers
User.objects.create_user(
    username='john',
    password='secret123'
)

# Django automatically hashes:
# 'secret123' → 'pbkdf2_sha256$260000$abc...xyz'
```

How hashing works:

```
Input: secret123
Add salt: secret123_randomsalt
Hash function: SHA256
Output: 64-character hex string

Stored in DB: pbkdf2_sha256$260000$salt$hash
```

Checking passwords:

```
user.check_password('secret123')  # Returns True/False
# Django:
# 1. Gets salt from stored hash
# 2. Hashes input with same salt
# 3. Compares hashes
```

Why secure:

- Can't reverse hash to get password
- Same password + different salt = different hash
- Takes time to compute (prevents brute force)

---

11. Common Errors & Solutions

11.1 CSRF Verification Failed

Error: `CSRF token from POST incorrect`

Why it happened:

- Frontend not sending CSRF token in header
- Cookie not being sent with request

Solution:

1. Ensure `withCredentials: true` in axios config
2. Add CSRF token to unsafe requests:

```
const csrfToken = getCsrfToken();
config.headers['X-CSRFToken'] = csrfToken;
```

3. Call `/api/auth/csrf/` on app load to get cookie

11.2 Third-Party Login Failure (OAuth)

Error: "An error occurred while attempting to login via your third-party account"

Why it happened:

- Incorrect allauth settings (deprecated variable names)
- Wrong Site domain configuration

Solution:

1. Updated settings to use correct allauth variables:

```
# Wrong (deprecated)
ACCOUNT_AUTHENTICATION_METHOD = 'email'

# Correct
ACCOUNT_LOGIN_METHODS = {'email'}
ACCOUNT_SIGNUP_FIELDS = ['email*', 'password1*', 'password2*']
```

2. Fixed Site object:

```
python manage.py fix_site
```

11.3 Cross-Site Cookie Warning

Error: "Mark cross-site cookies as Secure to allow setting them in cross-site contexts"

Why it happened:

- `SameSite=None` requires `Secure=True` (HTTPS)
- Development uses HTTP (no HTTPS)
- Django admin couldn't set session cookies

Solution: Changed to environment-aware cookie settings:

```
if DEBUG:
    # Development: Use Lax (works on HTTP)
    SESSION_COOKIE_SAMESITE = 'Lax'
    SESSION_COOKIE_SECURE = False
else:
    # Production: Use None with HTTPS
    SESSION_COOKIE_SAMESITE = 'None'
    SESSION_COOKIE_SECURE = True
```

11.4 "This email is already registered"

Not an error - this is **intended behavior**!

Why we check:

- Prevent duplicate accounts
- Tell user to use Google if they signed up with Google
- Guide users to correct login method

Flow:

```
def validate_email(self, value):
    if User.objects.filter(email__iexact=value).exists():
        user = User.objects.get(email__iexact=value)

        # Check if has Google account
        if SocialAccount.objects.filter(user=user, provider='google').exists():
            raise ValidationError(
                "This email is already registered with Google. "
                "Please sign in using 'Continue with Google'."
            )
        else:
            raise ValidationError(
                "This email is already registered. Please log in."
            )
```

11.5 ModuleNotFoundError

Error: `ModuleNotFoundError: No module named 'dj_database_url'`

Why it happened:

- Python packages not installed
- Virtual environment not activated

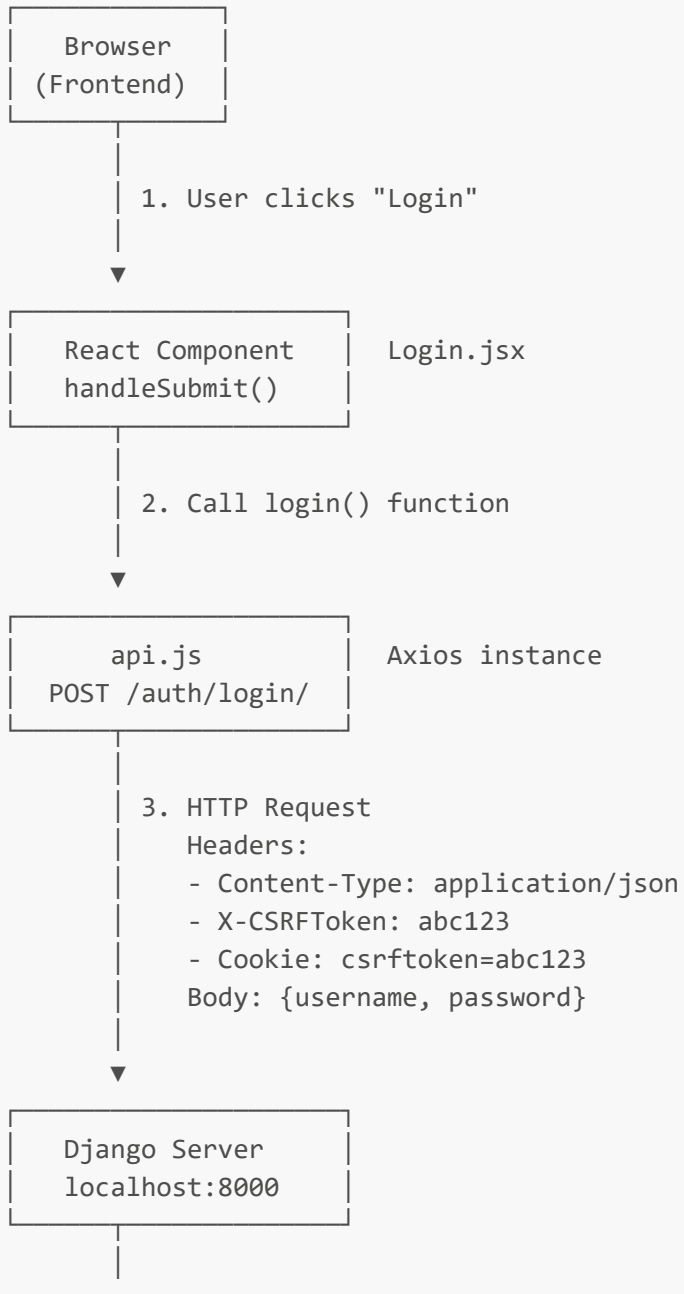
Solution:

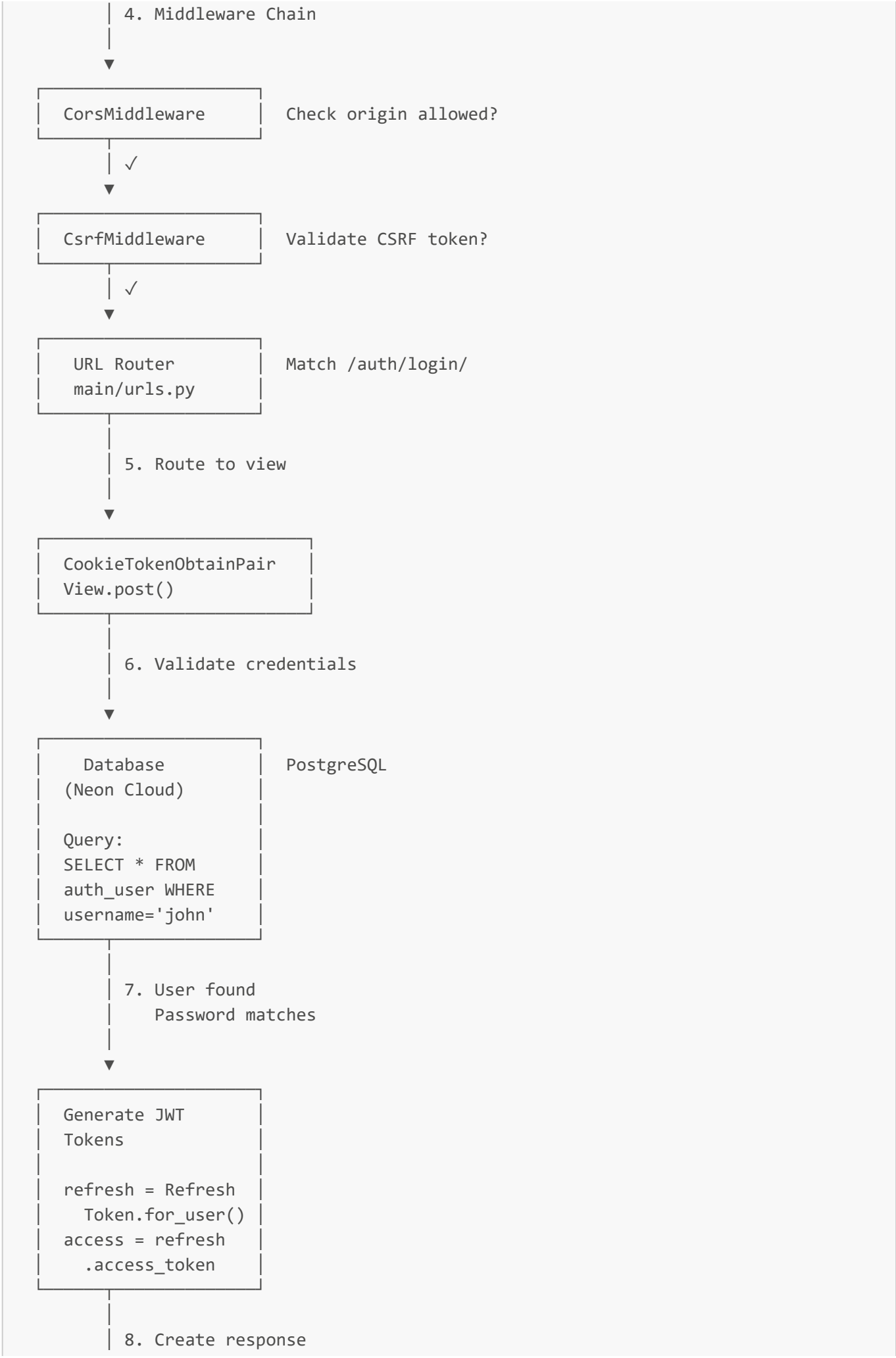
```
# Activate virtual environment
.\benv\Scripts\Activate.ps1

# Install dependencies
pip install -r requirements.txt
```

---

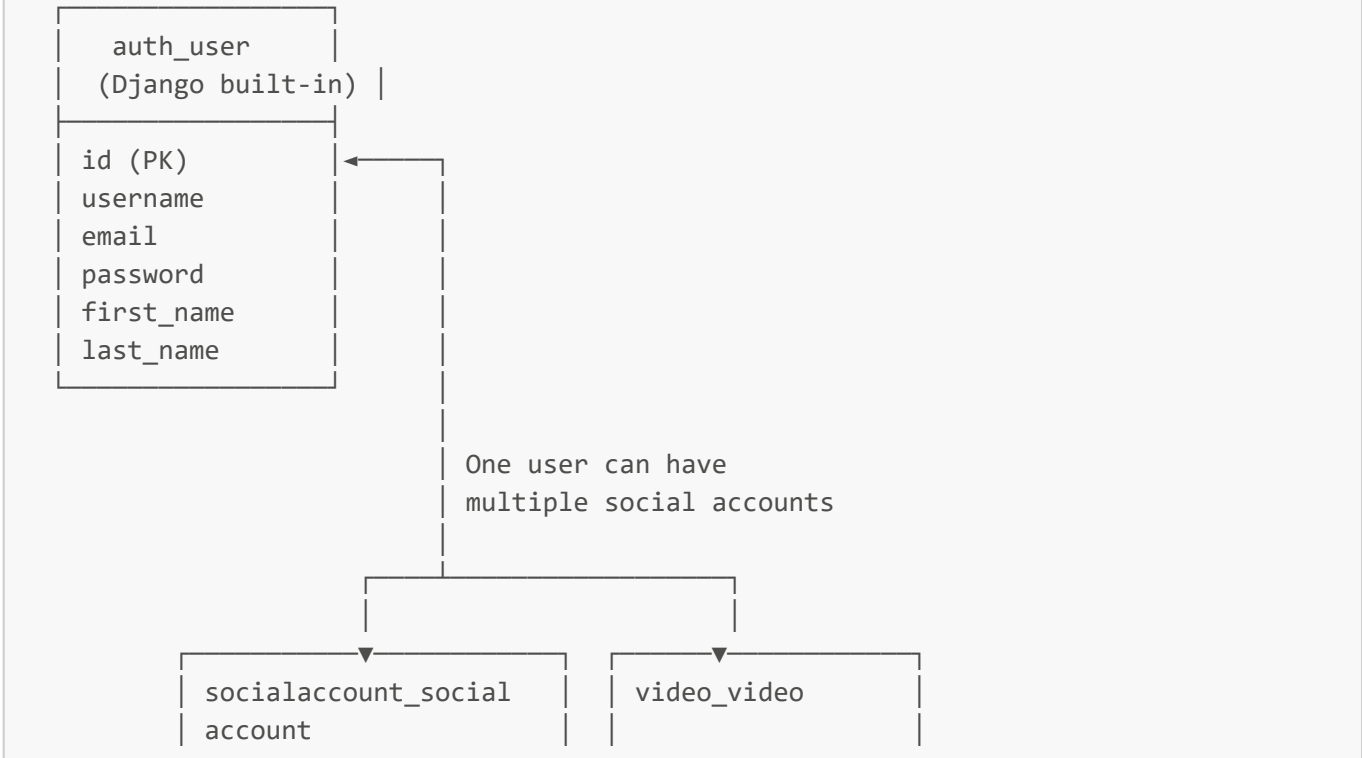
12. How Everything Connects

12.1 Complete Request Flow Diagram



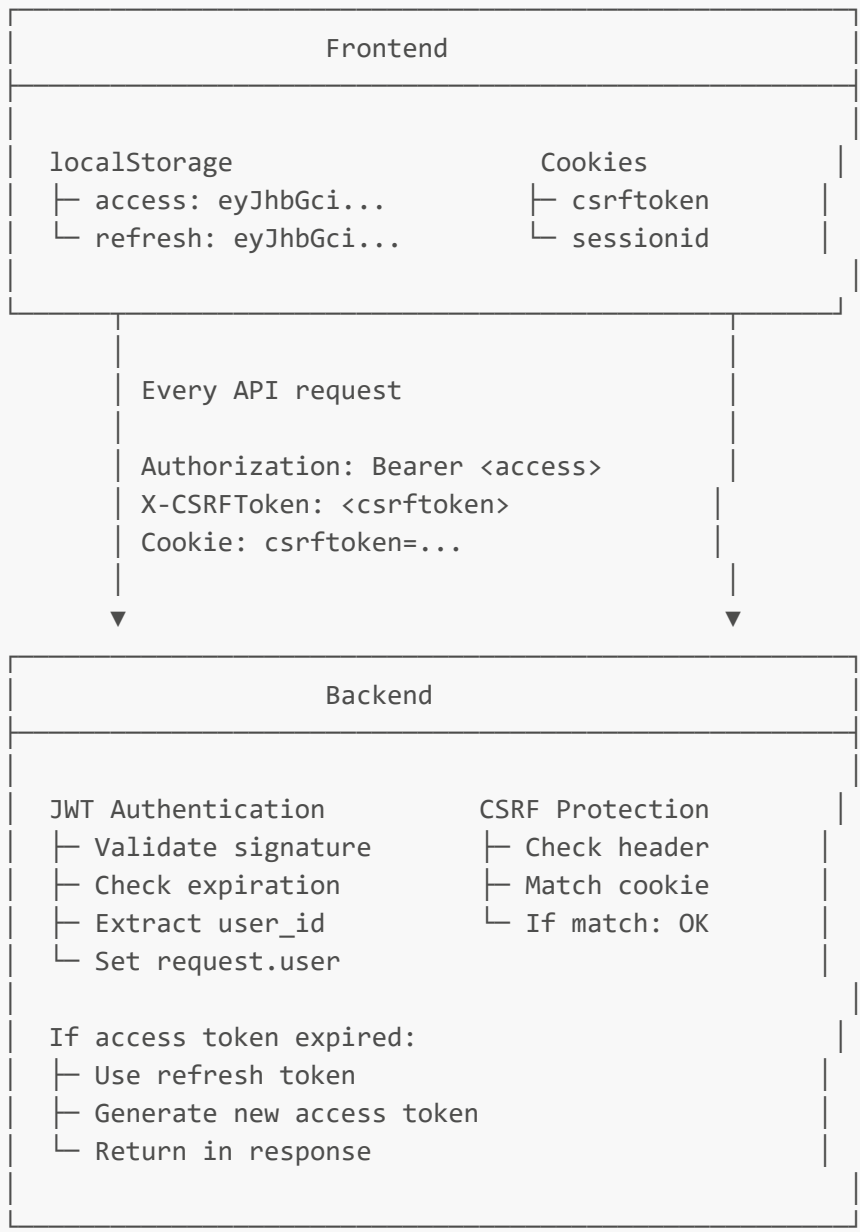

4. Middleware Chain
CorsMiddleware
Check origin allowed?
✓
CsrfMiddleware
Validate CSRF token?
✓
URL Router
main/urls.py
Match /auth/login/
5. Route to view
CookieTokenObtainPair
View.post()
6. Validate credentials
Database
(Neon Cloud)
PostgreSQL
Query:
SELECT * FROM
auth_user WHERE
username='john'
7. User found
Password matches
Generate JWT
Tokens
refresh = Refresh
Token.for_user()
access = refresh
.access_token
8. Create response

12.2 Database Relationships



```
┌──────────────────────────┐   ┌──────────────────────┐
│ id (PK)                  │   │ id (PK)              │
│ user_id (FK)             │   │ title                │
│ provider (google)        │   │ description          │
│ uid (Google ID)          │   │ video_url            │
│ extra_data (JSON)        │   │ thumbnail            │
└──────────────────────────┘   │ created_at           │
                               └──────────────────────┘
```

12.3 Token Flow



12.4 OAuth Account Linking

```
Scenario 1: User signs up with form, then uses Google
──────────────────────────────────────────────────────
```

```
1. User creates form account
   ├─ Email: john@gmail.com
   ├─ Password: secret123
   └─ Saved in: auth_user

2. Later, user clicks "Google"
   └─ Google provides: john@gmail.com

3. Adapter checks database
   ├─ Email exists? YES
   └─ Link Google account

4. Database now has:
   ├─ auth_user (id=1, email=john@gmail.com)
   └─ socialaccount_socialaccount (user_id=1, provider=google)

5. User can login with:
   ├─ Form: email + password
   └─ Google: just click button


Scenario 2: User signs up with Google, tries form signup
──────────────────────────────────────────────────────────

1. User clicks "Google"
   ├─ Google provides: jane@gmail.com
   └─ Creates user (no password)

2. Database has:
   ├─ auth_user (id=2, email=jane@gmail.com, password='!')
   │  Note: '!' means "no usable password"
   └─ socialaccount_socialaccount (user_id=2, provider=google)

3. Later, user tries form signup
   └─ Email: jane@gmail.com

4. Serializer validation:
   ├─ Email exists? YES
   ├─ Has Google account? YES
   └─ Return error: "Use Google to login"

5. User guidance:
   └─ "This email is registered with Google. Click 'Continue with Google'."
```

Summary

What We Built

A modern full-stack learning platform with:

- **Dual authentication**: Form-based and Google OAuth
- **RESTful API**: Django REST Framework
- **React frontend**: Modern UI with React hooks
- **Cloud database**: PostgreSQL on Neon
- **Secure tokens**: JWT-based authentication
- **Smart account linking**: Prevents duplicates, guides users

Key Technologies

Backend:

- Django 6.0 (Python web framework)
- Django REST Framework (API building)
- Django-allauth (OAuth integration)
- Simple JWT (Token authentication)
- PostgreSQL (Database)

Frontend:

- React (UI library)
- Vite (Build tool)
- Axios (HTTP client)

Why Each Component Exists

Serializers: Convert between Python/database and JSON **Views**: Handle HTTP requests, business logic **Models**: Define database structure **Middleware**: Process every request (CORS, CSRF, auth) **JWT Tokens**: Stateless authentication **CSRF Tokens**: Prevent cross-site attacks **OAuth Adapter**: Link social accounts to users **Cookies**: Store session and CSRF data **LocalStorage**: Store JWT tokens

Next Steps to Learn

1. **Add more models**: Courses, Progress, Comments
2. **Add relationships**: User enrollments, video completions
3. **Add permissions**: Teachers vs Students
4. **Add file uploads**: Profile pictures, course materials
5. **Add email**: Verification, password reset
6. **Add search**: Full-text search for courses
7. **Add caching**: Redis for performance
8. **Deploy**: Production hosting (AWS, Heroku, etc.)

Remember: This is a learning project. Every error you encounter teaches you something new. The architecture we built is production-ready and follows industry best practices.

Happy coding! 🚀