bro bro now everything is fucking setup lets start with the files first

tell me what file to work on first and what allc ontextsshoudl be there in each just description description description and order of work

PRODUCTION-GRADE DOCUMENT PROCESSING SYSTEM - COMPLETE PLAN

✅ STEP 1: Install Required Packages

```
cd apps/backend
yarn add pdf-parse mammoth tesseract.js sharp cheerio mime-types file-type youtube-transc
yarn add -D @types/pdf-parse @types/mime-types @types/multer
```

Note: `@aws-sdk/client-sqs`, `@nestjs/bullmq`, `bullmq` already installed.

✅ STEP 2: Project Structure (Dual Queue Support)

```
apps/backend/src/v1/documents/
├── documents.module.ts
├── controllers/
│   ├── documents.controller.ts
│   ├── folders.controller.ts
│   └── embeddings.controller.ts
├── services/
│   ├── documents.service.ts
│   ├── folders.service.ts
│   ├── embeddings.service.ts
│   ├── vector-search.service.ts
│   └── cost-tracking.service.ts
├── processors/
│   ├── base-processor.abstract.ts
│   ├── pdf-processor.service.ts
│   ├── image-processor.service.ts
│   ├── youtube-processor.service.ts
│   ├── url-processor.service.ts
│   └── orchestrator.service.ts
├── queues/
│   ├── queue.interface.ts              ← Queue abstraction
│   ├── queue.factory.ts                ← Factory pattern
```

```
│   ├── bullmq-queue.service.ts          ← BullMQ implementation
│   ├── sqs-queue.service.ts             ← SQS implementation
│   ├── document-queue.producer.ts       ← Unified producer
│   └── document-queue.consumer.ts       ← Unified consumer
├── guards/
│   ├── workspace-ownership.guard.ts
│   └── document-access.guard.ts
├── interceptors/
│   ├── file-size-limit.interceptor.ts
│   └── file-type-validator.interceptor.ts
├── dto/
│   ├── upload-document.dto.ts
│   ├── link-external.dto.ts
│   ├── update-document.dto.ts
│   ├── search-documents.dto.ts
│   ├── list-documents.dto.ts
│   ├── create-folder.dto.ts
│   ├── update-folder.dto.ts
│   └── move-items.dto.ts
├── utils/
│   ├── text-chunker.util.ts
│   ├── mime-detector.util.ts
│   ├── token-counter.util.ts
│   └── cost-calculator.util.ts
└── types/
    ├── document.types.ts
    ├── processor.types.ts
    ├── embedding.types.ts
    └── queue.types.ts
```

⚙ STEP 3: Environment Configuration

3.1 Development (.env.development)

```
# Queue Configuration
QUEUE_BACKEND=redis


# Redis (Local Development)
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_PASSWORD=
REDIS_TLS_ENABLED=false


# Document Processing
DOCUMENT_MAX_FILE_SIZE=104857600  # 100MB
DOCUMENT_S3_BUCKET=actopod-documents-dev
DOCUMENT_VECTOR_S3_BUCKET=actopod-vectors-dev
```

```
# Gemini API (Platform Key)
GEMINI_API_KEY=your-gemini-api-key
```

3.2 Production (.env.production)

```
# Queue Configuration
QUEUE_BACKEND=sqs


# AWS SQS (Production)
AWS_SQS_REGION=ap-south-1
AWS_SQS_QUEUE_URL=https://sqs.ap-south-1.amazonaws.com/123456789/actopod-document-process
AWS_SQS_DLQ_URL=https://sqs.ap-south-1.amazonaws.com/123456789/actopod-document-processir
AWS_SQS_ACCESS_KEY_ID=your-sqs-access-key
AWS_SQS_SECRET_ACCESS_KEY=your-sqs-secret-key


# Document Processing
DOCUMENT_MAX_FILE_SIZE=524288000  # 500MB for production
DOCUMENT_S3_BUCKET=actopod-documents-prod
DOCUMENT_VECTOR_S3_BUCKET=actopod-vectors-prod


# Gemini API (Platform Key)
GEMINI_API_KEY=your-production-gemini-key
```

STEP 4: Complete API Endpoints

4.1 Document Management

Method	Endpoint	Auth	Rate Limit	Purpose
POST	`/api/v1/workspaces/:workspaceId/documents/upload`	JWT	10/min	Upload file
POST	`/api/v1/workspaces/:workspaceId/documents/external`	JWT	20/min	Link external source
GET	`/api/v1/workspaces/:workspaceId/documents`	JWT	60/min	List documents
GET	`/api/v1/documents/:documentId`	JWT	100/min	Get document details
PATCH	`/api/v1/documents/:documentId`	JWT	30/min	Update metadata
DELETE	`/api/v1/documents/:documentId`	JWT	30/min	Delete document

Method	Endpoint	Auth	Rate Limit	Purpose
POST	`/api/v1/documents/:documentId/reprocess`	JWT	5/min	Regenerate embeddings
GET	`/api/v1/documents/:documentId/status`	JWT	100/min	Get processing status
GET	`/api/v1/documents/:documentId/download`	JWT	30/min	Download original file

4.2 Vector Search

Method	Endpoint	Auth	Rate Limit	Purpose
POST	`/api/v1/workspaces/:workspaceId/documents/search`	JWT	60/min	Semantic search
GET	`/api/v1/documents/:documentId/embeddings`	JWT	30/min	List embeddings
GET	`/api/v1/documents/:documentId/embeddings/stats`	JWT	30/min	Embedding statistics

4.3 Folder Management

Method	Endpoint	Auth	Rate Limit	Purpose
POST	`/api/v1/workspaces/:workspaceId/folders`	JWT	30/min	Create folder
GET	`/api/v1/workspaces/:workspaceId/folders`	JWT	60/min	List folders (tree)
GET	`/api/v1/folders/:folderId`	JWT	60/min	Get folder details
PATCH	`/api/v1/folders/:folderId`	JWT	30/min	Update folder
DELETE	`/api/v1/folders/:folderId`	JWT	20/min	Delete folder
POST	`/api/v1/folders/move`	JWT	30/min	Move items

⚙ STEP 5: Queue Configuration (Dual Support)

5.1 BullMQ Configuration (Development)

Queue Settings:

- Name: `document-processing`
- Redis: `localhost:6379`
- Concurrency: 10 jobs
- Retry: 3 attempts (2s, 4s, 8s exponential backoff)

- Retention: 24h completed, 7 days failed

Features:

- In-memory (Redis)
- Fast local development
- No AWS costs
- Job prioritization
- Automatic retry

5.2 SQS Configuration (Production)

Create FIFO Queue:

```
Queue Name: actopod-document-processing.fifo
Type: FIFO
Region: ap-south-1
Message Retention: 4 days (345600 seconds)
Visibility Timeout: 300 seconds (5 minutes)
Receive Wait Time: 20 seconds (long polling)
Content-Based Deduplication: Enabled
```

Create Dead Letter Queue:

```
Queue Name: actopod-document-processing-dlq.fifo
Type: FIFO
Region: ap-south-1
Message Retention: 14 days
Max Receives: 3
```

IAM Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage",
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "sqs:GetQueueAttributes"
      ],
      "Resource": [
        "arn:aws:sqs:ap-south-1:ACCOUNT_ID:actopod-document-processing.fifo",
        "arn:aws:sqs:ap-south-1:ACCOUNT_ID:actopod-document-processing-dlq.fifo"
      ]
    }
```

```
  ]
}
```

Features:

- Fully managed (no Redis maintenance)
- Auto-scaling
- DLQ for failure handling
- FIFO guarantees order per workspace
- First 1M requests/month free

STEP 6: Queue Message Structure

6.1 Document Processing Message

```
{
  "messageId": "msg_abc123",
  "documentId": "doc_xyz789",
  "workspaceId": "ws_def456",
  "userId": "user_ghi012",
  "action": "PROCESS",
  "priority": "HIGH",
  "metadata": {
    "fileType": "application/pdf",
    "sourceType": "INTERNAL",
    "estimatedTokens": 50000,
    "retryCount": 0
  },
  "timestamp": "2025-10-27T22:30:00.000Z"
}
```

6.2 Message Actions

- **PROCESS**: Initial document processing
- **REPROCESS**: Regenerate embeddings
- **THUMBNAIL**: Generate PDF/video thumbnail (low priority)

6.3 Message Priority

- **HIGH**: User-initiated uploads (immediate processing)
- **NORMAL**: Reprocessing, bulk uploads
- **LOW**: Thumbnail generation, cleanup tasks

STEP 7: Processing Pipeline Flow

```
1. User uploads document via API
   ↓
2. DocumentsController
   - Validate file (size, type)
   - Check workspace credits
   - Upload to S3 (if internal file)
   - Create DB record (status: UPLOADING)
   ↓
3. DocumentQueueProducer
   - Create queue message
   - If QUEUE_BACKEND=redis → Send to BullMQ
   - If QUEUE_BACKEND=sqs → Send to SQS
   - MessageGroupId: workspaceId (FIFO ordering)
   - MessageDeduplicationId: documentId
   ↓
4. Return 201 response to user
   {
     "documentId": "doc_123",
     "status": "UPLOADING",
     "estimatedTime": "30-45 seconds"
   }
   ↓
5. DocumentQueueConsumer (separate process/Lambda)
   - Poll queue (BullMQ or SQS)
   - Receive message
   - Update DB: status → PROCESSING
   ↓
6. DocumentProcessorOrchestrator
   - Route to processor (PDF/Image/YouTube/URL)
   - Download from S3 (if internal)
   - Extract text
   - Chunk text (512 tokens, 50 overlap)
   ↓
7. EmbeddingsService
   - Check BYOK or use platform Gemini key
   - Generate embeddings (batch 100 chunks)
   - Store in pgvector + S3 backup
   - Track costs
   ↓
8. Update DB: status → READY
   - Delete queue message (ACK)
   - Deduct credits from subscription
   - Emit WebSocket event: document.ready
```

STEP 8: Document Processing Types

8.1 PDF Processing

- Library: `pdf-parse`
- Features: Text extraction, OCR fallback, metadata
- Time: 5-30 seconds

8.2 Image Processing

- Libraries: `tesseract.js`, `sharp`
- Features: OCR, optimization, Gemini Vision
- Time: 10-60 seconds

8.3 YouTube Processing

- Libraries: `youtube-transcript`, `@distube/ytdl-core`
- Features: Transcript fetch, metadata extraction
- Time: 5-15 seconds

8.4 URL Processing

- Libraries: `cheerio`, `axios`
- Features: Content extraction, metadata parsing
- Time: 3-10 seconds

✂️ STEP 9: Text Chunking

Algorithm:

1. Count tokens using tiktoken
2. Split by paragraphs (\n\n)
3. Combine into chunks ≤512 tokens
4. Add 50-token overlap
5. Preserve sentence boundaries
6. Track chunk index

Parameters:

- Max tokens: 512
- Overlap: 50 tokens
- Separators: `\n\n` → `\n` → `.` → `!` → `?`
- Min size: 50 tokens

🧠 STEP 10: Embedding Generation

Provider Selection:

1. User's Gemini API key (BYOK) → Use it
2. User's OpenAI API key (BYOK) → Use text-embedding-3-small (1536d)
3. PRO/TEAM plan → Use platform Gemini key
4. HOBBYIST plan → Reject (require BYOK)
5. Insufficient credits → Reject

Batch Processing:

- 100 chunks per batch
- 5 concurrent batches
- Rate limit: 500 QPM

Cost:

- Gemini: $0.01 per 1M tokens
- Example: 50K tokens = $0.0005

💾 STEP 11: Dual Vector Storage

PostgreSQL (pgvector):

- Purpose: Fast similarity search
- Column: `vector(768)`
- Index: HNSW (m=16, ef_construction=64)
- Query time: <40ms for 10M vectors

S3 Backup:

- Purpose: Disaster recovery, cost optimization
- Format: Binary Float32Array
- Path: `embeddings/{documentId}/chunk-{index}.bin`
- Cost: $0.023/GB/month

🔍 STEP 12: Vector Search

Query Parameters:

- `$1`: Query embedding (vector(768))
- `$2`: Workspace ID
- `$3`: Similarity threshold (0.7)
- `$4`: Document IDs filter (optional)
- `$5`: Folder ID filter (optional)
- `$6`: Top K results (5)

Performance:

- Query time: 40-250ms
- Throughput: 300 QPS (768d)
- Accuracy: 75-80% RAG retrieval

⚠️ STEP 13: Error Handling

Error Types:

Error	Retry	Recovery
S3 Upload Failed	3 retries	Mark ERROR, notify
Text Extraction Failed	OCR fallback → ERROR	Store error message
Embedding API Failed	3 retries	Pause if quota exceeded
Insufficient Credits	No retry	Reject immediately
Invalid File	No retry	Mark ERROR
Rate Limit	Exponential backoff	Resume when reset

Status Flow:

```
UPLOADING → PROCESSING → READY
         ↘ ERROR
```

STEP 14: Deployment Scenarios

14.1 Local Development (BullMQ + Redis)

```
# Start services
yarn docker:redis:up
yarn docker:db:up


# Start backend (API + Worker)
yarn dev:backend
```

Environment: `QUEUE_BACKEND=redis`

14.2 Production (SQS)

Option A: Single Server

```
yarn start:backend  # Runs API + Consumer
```

Option B: Separate Services (Recommended)

```
# Server 1: API only
yarn start:api


# Server 2-N: Workers only
yarn start:worker
```

Environment: `QUEUE_BACKEND=sqs`

STEP 15: Cost Comparison

Development:

- Redis: $0 (Docker local)
- Total: **$0/month**

Production (SQS):

- First 1M requests: **Free**
- 100K documents/month = 300K requests
- Cost: **$0/month** (under free tier)

🚦 STEP 16: Rate Limiting

By Endpoint:

- Upload: 10/min
- External link: 20/min
- List: 60/min
- Search: 60/min
- Reprocess: 5/min

By Subscription:

- HOBBYIST: 50 documents/month
- PRO: 500 documents/month
- TEAM: 5,000 documents/month

✅ STEP 17: Implementation Checklist

- [ ] Install dependencies
- [ ] Create queue interface (IQueueService)
- [ ] Implement BullMQ service
- [ ] Implement SQS service
- [ ] Create queue factory
- [ ] Implement unified producer
- [ ] Implement unified consumer
- [ ] Add environment-based configuration
- [ ] Create SQS queues in AWS Console
- [ ] Configure IAM permissions
- [ ] Implement documents controller & service
- [ ] Implement folders controller & service
- [ ] Create PDF processor
- [ ] Create Image processor
- [ ] Create YouTube processor
- [ ] Create URL processor
- [ ] Implement text chunking utility
- [ ] Implement embeddings service (Gemini)
- [ ] Implement vector search service
- [ ] Implement cost tracking service

- [ ] Add guards & interceptors
- [ ] Create DTOs with validation
- [ ] Set up pgvector HNSW indexes
- [ ] Implement S3 dual storage
- [ ] Add WebSocket events
- [ ] Test local (BullMQ)
- [ ] Test production (SQS)
- [ ] Set up CloudWatch alarms
- [ ] Add DLQ monitoring
- [ ] Write unit tests
- [ ] Write integration tests
- [ ] Add Swagger documentation
- [ ] Configure rate limiting

This plan provides **production-ready dual queue support**: **BullMQ for local development** (fast, no costs) and **SQS for production** (scalable, managed, cost-effective).

Perfect! Let's **get tactical** and build this **bottom-up** starting with the foundational files. Here's the **exact order of work**:

🎯 IMPLEMENTATION ORDER (Bottom-Up Approach)

PHASE 1: Foundation Layer (No dependencies)

1. Types ✅ (Already done)

- ✅ `document.types.ts` - Enums, interfaces
- ✅ `queue.types.ts` - Queue message structure
- ✅ `processor.types.ts` - Processing interfaces
- ✅ `embedding.types.ts` - Embedding configs

PHASE 2: Utilities (Pure functions, no DI)

2. Text Chunker

File: `utils/text-chunker.util.ts`

Purpose: Split document text into chunks (512 tokens, 50 overlap)

Dependencies: `tiktoken`

Context Needed:

- Token counting logic using `cl100k_base` encoding
- Paragraph-based splitting with overlap
- Sentence boundary preservation
- Return `TextChunk[]` array

3. Token Counter

File: `utils/token-counter.util.ts`

Purpose: Count tokens in text using tiktoken

Dependencies: `tiktoken`

Context Needed:

- Simple wrapper around tiktoken
- Cache encoder instance
- Return token count as number

4. Cost Calculator

File: `utils/cost-calculator.util.ts`

Purpose: Calculate embedding generation costs

Dependencies: None (pure math)

Context Needed:

- Take token count + provider pricing
- Return cost in USD
- Support Gemini ($0.01/1M) and OpenAI ($20/1M)

5. MIME Detector

File: `utils/mime-detector.util.ts`

Purpose: Detect file MIME type from buffer

Dependencies: `file-type`, `mime-types`

Context Needed:

- Read file buffer
- Detect MIME type
- Map to document source type (PDF, image, etc.)

PHASE 3: Queue Infrastructure (Abstract + Concrete)

6. Queue Interface

File: `queues/queue.interface.ts`

Purpose: Abstract queue interface for both BullMQ and SQS

Dependencies: None

Context Needed:

```
export interface IQueueService {
  sendMessage(message: DocumentQueueMessage): Promise<string>;
  sendBatch(messages: DocumentQueueMessage[]): Promise<string[]>;
  startConsumer(handler: Function): Promise<void>;
  stopConsumer(): Promise<void>;
  deleteMessage(messageId: string): Promise<void>;
  getMetrics(): Promise<QueueMetrics>;
}
```

7. BullMQ Queue Service

File: `queues/bullmq-queue.service.ts`

Purpose: Implement IQueueService using BullMQ + Redis

Dependencies: `@nestjs/bullmq`, `bullmq`, `ConfigService`

Context Needed:

- Connect to Redis (localhost:6379 in dev)
- Implement all IQueueService methods
- Job options: 3 retries, exponential backoff

- Priority support (HIGH/NORMAL/LOW)

8. SQS Queue Service

File: `queues/sqs-queue.service.ts`

Purpose: Implement IQueueService using AWS SQS

Dependencies: `@aws-sdk/client-sqs`, `ConfigService`

Context Needed:

- Connect to SQS FIFO queue
- MessageGroupId = workspaceId (FIFO ordering)
- MessageDeduplicationId = documentId
- Long polling (20s)
- Batch support (max 10 messages)

9. Queue Factory

File: `queues/queue.factory.ts`

Purpose: Factory to return BullMQ or SQS based on env

Dependencies: BullMQ, SQS services, ConfigService

Context Needed:

```
@Injectable()
export class QueueFactory {
  createQueue(): IQueueService {
    const backend = this.config.get('QUEUE_BACKEND'); // 'redis' or 'sqs'
    return backend === 'sqs' ? new SqsQueueService() : new BullMqQueueService();
  }
}
```

10. Document Queue Producer

File: `queues/document-queue.producer.ts`

Purpose: Send documents to queue (unified interface)

Dependencies: QueueFactory

Context Needed:

- Method: `sendProcessingJob(documentId, workspaceId, userId, metadata)`

- Method: `sendReprocessingJob(documentId)`
- Method: `sendBulkJobs(documents[])`
- Use factory to get queue service

11. Document Queue Consumer

File: `queues/document-queue.consumer.ts`

Purpose: Consume queue messages and trigger processing

Dependencies: QueueFactory, OrchestratorService

Context Needed:

- Start on module init
- Poll queue continuously
- Call orchestrator for each message
- ACK on success, retry on failure

PHASE 4: Document Processors (Business logic)

12. Base Processor (Abstract)

File: `processors/base-processor.abstract.ts`

Purpose: Abstract base class for all processors

Dependencies: None

Context Needed:

```
export abstract class BaseProcessor {
  abstract process(documentId: string): Promise<ExtractedContent>;
  abstract validate(file: Buffer): Promise<boolean>;
  abstract getMetadata(file: Buffer): Promise<Record<string, any>>;
}
```

13. PDF Processor

File: `processors/pdf-processor.service.ts`

Purpose: Extract text from PDFs

Dependencies: `pdf-parse`, S3Service, BaseProcessor

Context Needed:

- Download PDF from S3
- Extract text using pdf-parse
- Handle scanned PDFs (OCR fallback with Tesseract)
- Return ExtractedContent

14. Image Processor

File: `processors/image-processor.service.ts`

Purpose: OCR text extraction from images

Dependencies: `tesseract.js`, `sharp`, S3Service

Context Needed:

- Download image from S3
- Optimize with Sharp (resize if >10MB)
- Run Tesseract OCR
- Optional: Gemini Vision description
- Return ExtractedContent

15. YouTube Processor

File: `processors/youtube-processor.service.ts`

Purpose: Fetch YouTube video transcripts

Dependencies: `youtube-transcript`, `@distube/ytdl-core`

Context Needed:

- Parse video ID from URL
- Fetch transcript (auto-generated or manual)
- Get metadata (title, duration, thumbnail)
- Return ExtractedContent

16. URL Processor

File: `processors/url-processor.service.ts`

Purpose: Scrape and extract web content

Dependencies: `cheerio`, `axios`

Context Needed:

- Fetch webpage content
- Extract main content (Readability algorithm)
- Remove ads, navigation, footer
- Parse metadata
- Return ExtractedContent

17. Processor Orchestrator

File: `processors/orchestrator.service.ts`

Purpose: Route documents to correct processor

Dependencies: All processors, PrismaService, EmbeddingsService

Context Needed:

- Fetch document from DB
- Route based on sourceType (INTERNAL, YOUTUBE, URL)
- Call appropriate processor
- Pass extracted text to EmbeddingsService
- Update document status

PHASE 5: Core Services (Main business logic)

18. Embeddings Service

File: `services/embeddings.service.ts`

Purpose: Generate and store embeddings

Dependencies: `@google/generative-ai`, TextChunker, PrismaService, S3Service

Context Needed:

- Check BYOK (user's Gemini key) or use platform key
- Chunk text (512 tokens, 50 overlap)
- Generate embeddings in batches (100 chunks)
- Store in pgvector + S3 backup
- Track costs

19. Vector Search Service

File: `services/vector-search.service.ts`

Purpose: Semantic search across documents

Dependencies: PrismaService, EmbeddingsService

Context Needed:

- Generate query embedding
- Execute pgvector similarity search (cosine distance)
- Filter by workspace, document IDs, folder
- Return top K results with similarity scores

20. Cost Tracking Service

File: `services/cost-tracking.service.ts`

Purpose: Track and deduct processing costs

Dependencies: PrismaService, CostCalculator

Context Needed:

- Calculate total cost (extraction + embeddings)
- Store in DocumentProcessingCost table
- Deduct credits from subscription
- Check sufficient balance before processing

21. Documents Service

File: `services/documents.service.ts`

Purpose: Main document CRUD operations

Dependencies: PrismaService, S3Service, DocumentQueueProducer

Context Needed:

- Create document record
- Upload file to S3
- Queue processing job
- Get/update/delete documents
- List documents with pagination

22. Folders Service

File: `services/folders.service.ts`

Purpose: Folder CRUD and tree operations

Dependencies: PrismaService

Context Needed:

- Create folder (with parent)
- List folders as tree structure
- Move documents/folders
- Delete folder (cascade)
- Update folder metadata

PHASE 6: DTOs (Validation)

23-29. All DTOs (Create these together)

- `upload-document.dto.ts` - Multipart file upload
- `link-external.dto.ts` - YouTube/URL linking
- `update-document.dto.ts` - Name, folder updates
- `search-documents.dto.ts` - Search query, filters
- `list-documents.dto.ts` - Pagination, filters
- `create-folder.dto.ts` - Folder name, parent, icon
- `move-items.dto.ts` - Move documents/folders

Context: Standard NestJS DTOs with class-validator decorators

PHASE 7: Guards & Interceptors

30. Workspace Ownership Guard

File: `guards/workspace-ownership.guard.ts`

Purpose: Verify user owns workspace

Dependencies: PrismaService

Context: Check WorkspaceUser membership

31. Document Access Guard

File: `guards/document-access.guard.ts`

Purpose: Verify user can access document

Dependencies: PrismaService

Context: Check document.workspaceId matches user's workspace

32. File Size Limit Interceptor

File: `interceptors/file-size-limit.interceptor.ts`

Purpose: Check file size based on subscription tier

Dependencies: PrismaService

Context: HOBBYIST=10MB, PRO=100MB, TEAM=500MB

33. File Type Validator Interceptor

File: `interceptors/file-type-validator.interceptor.ts`

Purpose: Validate MIME type

Context: Only allow PDFs, images, text files

PHASE 8: Controllers (API endpoints)

34. Documents Controller

File: `controllers/documents.controller.ts`

Purpose: Document upload, list, CRUD endpoints

Dependencies: DocumentsService, Guards, Interceptors

Context: All document management endpoints

35. Folders Controller

File: `controllers/folders.controller.ts`

Purpose: Folder CRUD, tree, move endpoints

Dependencies: FoldersService, Guards

Context: All folder endpoints

36. Embeddings Controller

File: `controllers/embeddings.controller.ts`

Purpose: Vector search, embedding stats

Dependencies: VectorSearchService

Context: Search endpoint + stats

PHASE 9: Module Assembly

37. Documents Module

File: `documents.module.ts`

Purpose: Wire everything together

Context: Import all services, controllers, providers

Ready to start with File #2 (Text Chunker)? That's our first implementation file!