File Storage & Media API (S3-Like) Backend Documentation

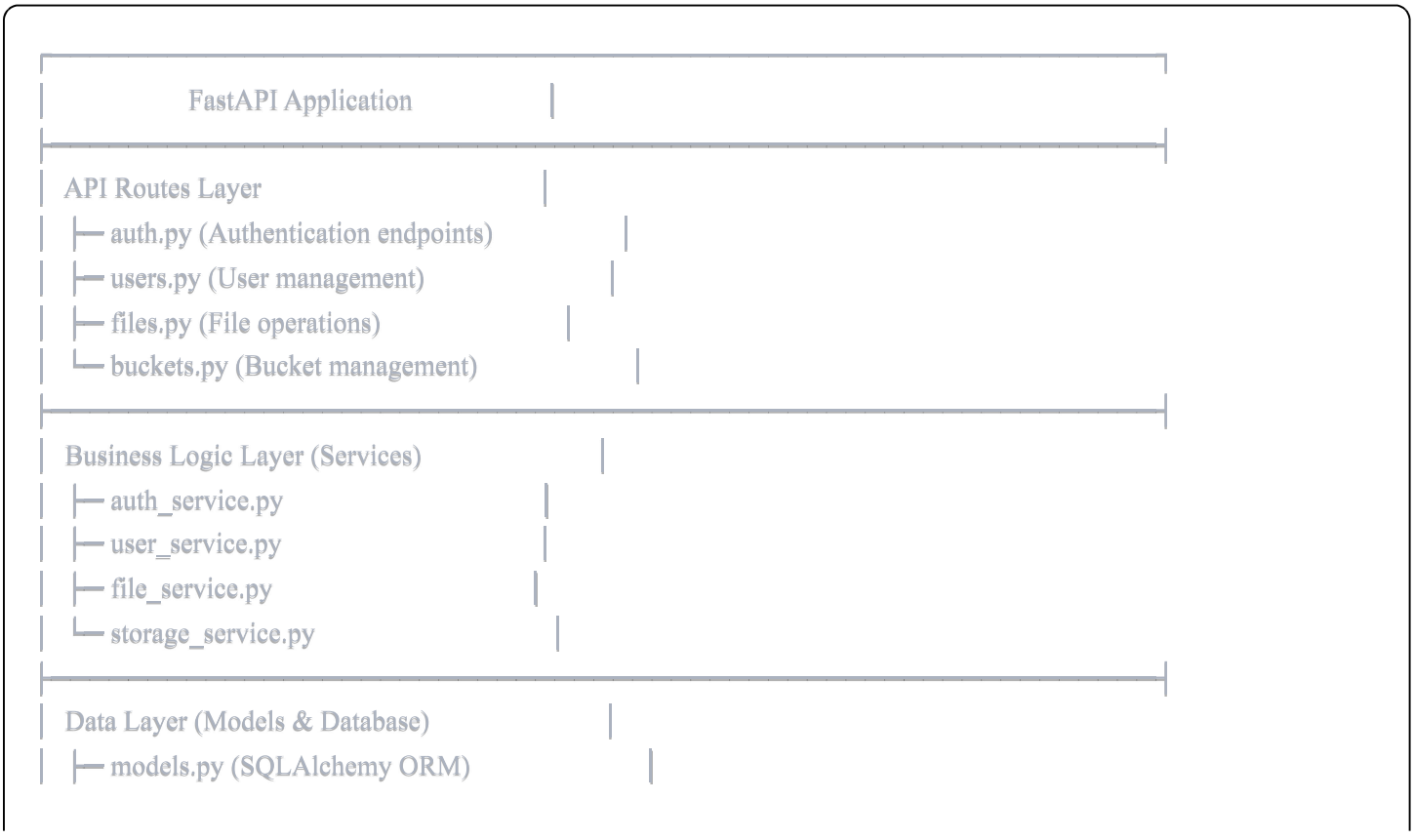
Project Overview

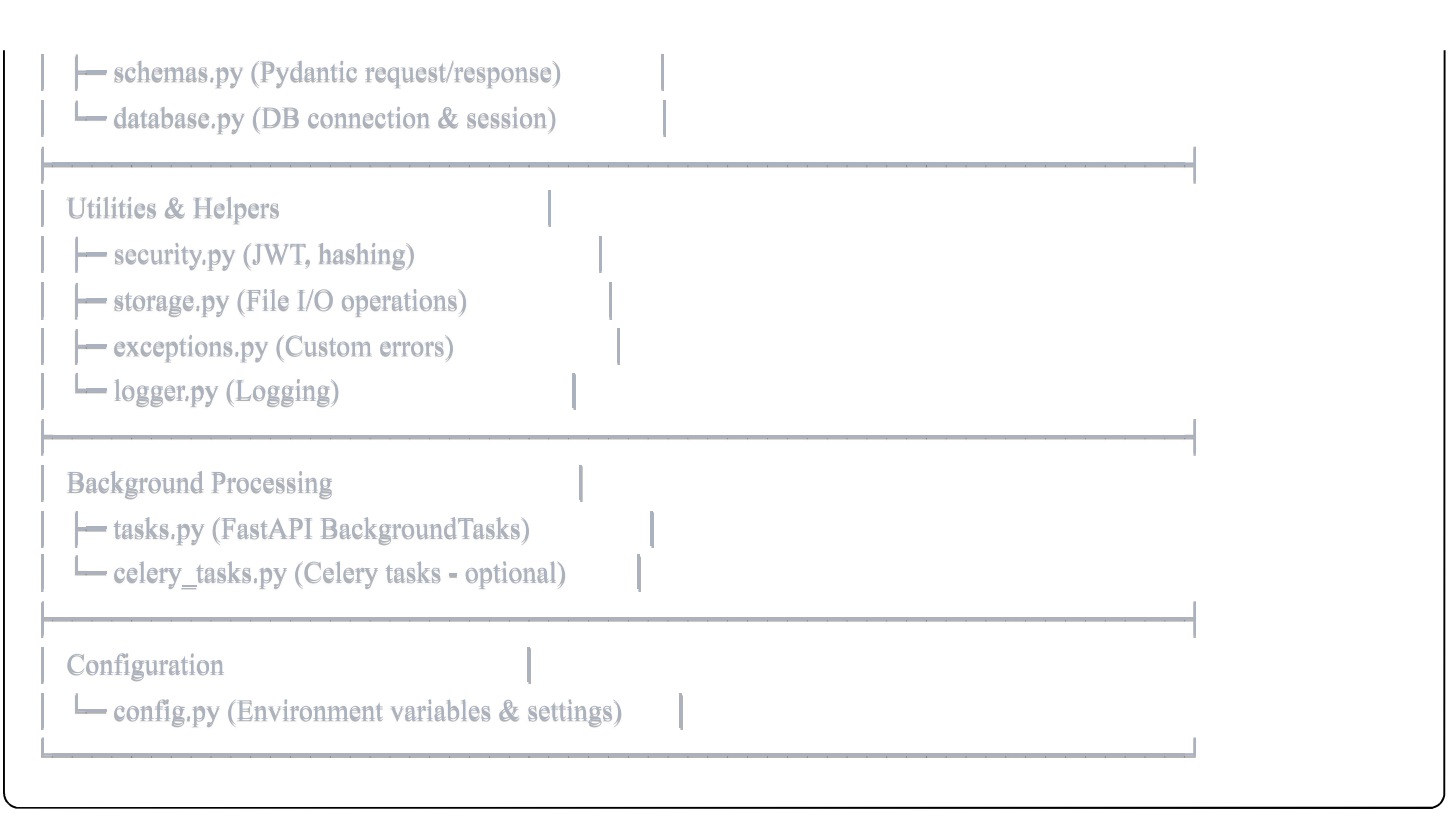
This is a comprehensive backend framework for building an S3-like file storage and media API using FastAPI, PostgreSQL, and JWT authentication. The system supports file uploads, downloads, access control, and background processing with a modular architecture for scalability.

Tech Stack:

- Framework: FastAPI (async/await)
- Database: PostgreSQL (production) / SQLite (testing)
- Authentication: JWT with Argon2 password hashing
- Storage: Local filesystem (upgradable to AWS S3)
- Background Tasks: FastAPI BackgroundTasks / Celery
- Testing: pytest, httpx
- API Documentation: OpenAPI/Swagger

Architecture Overview





Module Breakdown & Learning Path

Module 1: Project Setup & Configuration

Purpose: Initialize project structure, environment management, and database configuration.

Files to Create:

- `config.py` - Application settings
- `main.py` - FastAPI app initialization
- `requirements.txt` - Dependencies
- `.env.example` - Environment variables template
- `alembic/` - Database migrations

Key Concepts to Learn:

- Python virtual environments and dependency management
- Pydantic Settings for configuration management
- Environment variables and `.env` files
- FastAPI application factory pattern
- SQLAlchemy engine and session management

- Alembic for database migrations

Tools & Libraries:

```
fastapi==0.104.1
uvicorn==0.24.0
sqlalchemy==2.0.23
pydantic==2.5.0
pydantic-settings==2.1.0
python-dotenv==1.0.0
alembic==1.12.1
psycopg2-binary==2.9.9 (PostgreSQL driver)
```

Learning Resources:

- FastAPI official documentation on application creation
- Pydantic settings documentation
- SQLAlchemy 2.0 Core documentation
- Alembic migrations guide
- 12 Factor App methodology

Tasks to Complete:

1. Create project directory structure
2. Set up Python virtual environment
3. Install and configure dependencies
4. Create configuration file with environment variable management
5. Set up database connection strings (PostgreSQL and SQLite)
6. Initialize SQLAlchemy engine and session factory
7. Create initial database migration template

Module 2: Authentication & Security

Purpose: Implement JWT-based authentication with secure password hashing.

Files to Create:

- `app/core/security.py` - JWT and password hashing utilities
- `app/core/exceptions.py` - Custom authentication exceptions
- `app/api/routes/auth.py` - Authentication endpoints
- `app/services/auth_service.py` - Authentication business logic
- `tests/test_auth.py` - Authentication tests

Key Concepts to Learn:

- JWT (JSON Web Tokens) structure and claims
- Password hashing with Argon2
- Access tokens vs refresh tokens
- FastAPI security dependencies
- Cookie vs Bearer token authentication
- Token expiration and refresh mechanisms
- Password validation rules

Tools & Libraries:

```
python-jose==3.3.0 (JWT handling)
python-multipart==0.0.6
passlib[argon2]==1.7.4 (Argon2 hashing)
cryptography==41.0.7
```

Learning Resources:

- OWASP password storage guidelines
- JWT best practices
- FastAPI security documentation
- Passlib Argon2 documentation
- Authentication flow diagrams

Tasks to Complete:

1. Implement password hashing with Argon2
2. Create JWT token generation and validation functions

3. Implement token refresh mechanism
4. Create FastAPI security dependency for route protection
5. Build login endpoint with credentials validation
6. Build token refresh endpoint
7. Build logout endpoint (token blacklisting - optional)
8. Write comprehensive authentication tests

Module 3: Database Models & ORM

Purpose: Define data models for users, buckets, files, and relationships using SQLAlchemy ORM.

Files to Create:

- `app/models/user.py` - User model
- `app/models/bucket.py` - Bucket model
- `app/models/file.py` - File model
- `app/models/base.py` - Base model with common fields
- `app/core/database.py` - Database session management
- `alembic/versions/` - Migration files

Key Concepts to Learn:

- SQLAlchemy ORM fundamentals
- Data types and column constraints
- Relationships (One-to-Many, Many-to-Many)
- Indexes and constraints
- Timestamps (created_at, updated_at)
- Database migrations with Alembic
- Soft deletes pattern
- Data integrity and foreign keys

Tools & Libraries:

```
sqlalchemy==2.0.23
alembic==1.12.1
sqlalchemy-utils==0.41.1
```

Learning Resources:

- SQLAlchemy ORM documentation
- Database schema design principles
- Normalization concepts
- Alembic migration guide
- PostgreSQL data types

Database Schema Overview:

Users Table:
- id (UUID, PK)
- username (VARCHAR, UNIQUE)
- email (VARCHAR, UNIQUE)
- password_hash (VARCHAR)
- is_active (BOOLEAN)
- created_at (TIMESTAMP)
- updated_at (TIMESTAMP)

Buckets Table:
- id (UUID, PK)
- owner_id (UUID, FK → Users)
- name (VARCHAR)
- is_public (BOOLEAN)
- storage_quota (BIGINT)
- created_at (TIMESTAMP)
- updated_at (TIMESTAMP)

Files Table:
- id (UUID, PK)
- bucket_id (UUID, FK → Buckets)
- owner_id (UUID, FK → Users)
- filename (VARCHAR)
- file_path (VARCHAR)
- file_size (BIGINT)
- mime_type (VARCHAR)
- checksum (VARCHAR)
- is_deleted (BOOLEAN)
- created_at (TIMESTAMP)
- updated_at (TIMESTAMP)

Access Control Table:
- id (UUID, PK)
- resource_id (UUID)
- resource_type (VARCHAR) - 'bucket' or 'file'
- user_id (UUID, FK → Users)
- permission (VARCHAR) - 'read', 'write', 'admin'
- created_at (TIMESTAMP)

Tasks to Complete:

1. Create base model with common fields

2. Create User model with proper validation
3. Create Bucket model with ownership tracking
4. Create File model with storage metadata
5. Create Access Control model for permissions
6. Define relationships between models
7. Create and execute database migrations
8. Write model validation tests

Module 4: Pydantic Schemas & Validation

Purpose: Define request and response schemas with validation using Pydantic V2.

Files to Create:

- `app/schemas/user.py` - User request/response schemas
- `app/schemas/bucket.py` - Bucket schemas
- `app/schemas/file.py` - File schemas
- `app/schemas/common.py` - Shared schemas
- `tests/test_schemas.py` - Schema validation tests

Key Concepts to Learn:

- Pydantic V2 model definition
- Field validation and constraints
- Custom validators
- Request and response separation
- Pagination schemas
- Error response formats
- OpenAPI schema generation
- Recursive and nested models

Tools & Libraries:

```
pydantic==2.5.0
pydantic[email]==2.5.0
```

Learning Resources:

- Pydantic V2 documentation
- Validation decorator usage
- Field types and constraints
- Custom type definitions
- OpenAPI schema documentation

Key Schemas:

```
UserCreate:
- username (str, min_length=3, max_length=50)
- email (EmailStr)
- password (str, min_length=8)

UserResponse:
- id (UUID)
- username (str)
- email (str)
- is_active (bool)
- created_at (datetime)

BucketCreate:
- name (str)
- is_public (bool, default=False)

FileResponse:
- id (UUID)
- filename (str)
- file_size (int)
- mime_type (str)
- created_at (datetime)
- url (str)

PaginatedResponse[T]:
- items (list[T])
- total (int)
- page (int)
- per_page (int)
```

Tasks to Complete:

1. Create user schemas with password validation
2. Create bucket schemas with naming rules
3. Create file metadata schemas
4. Implement custom validators for business rules
5. Create pagination schema
6. Create error response schemas
7. Test all schemas with valid/invalid data

Module 5: Storage & File Management

Purpose: Implement file upload, download, and storage operations.

Files to Create:

- `app/core/storage.py` - Storage abstraction layer
- `app/services/storage_service.py` - Storage business logic
- `app/services/file_service.py` - File service (database + storage)
- `tests/test_storage.py` - Storage operation tests

Key Concepts to Learn:

- File upload handling in FastAPI
- Streaming downloads for large files
- File path management and security
- Checksum/hash calculation (MD5, SHA256)
- File type validation
- Storage quota management
- Disk space monitoring
- S3 API compatibility design
- Temporary file cleanup

Tools & Libraries:

```
aiofiles==23.2.1 (Async file I/O)
python-multipart==0.0.6
hashlib (Built-in for checksums)
mimetypes (Built-in for MIME types)
shutil (Built-in for directory operations)
```

Learning Resources:

- FastAPI file upload documentation
- Streaming responses in FastAPI

- Python pathlib for safe path handling
- File system best practices
- Checksum verification guide

File Storage Structure:

```
storage/
├── bucket_uuid_1/
│   ├── file_uuid_1/metadata.json
│   ├── file_uuid_1/data
│   ├── file_uuid_2/metadata.json
│   └── file_uuid_2/data
└── bucket_uuid_2/
    └── ...
```

Tasks to Complete:

1. Create storage abstraction with local filesystem implementation
2. Implement file upload with multipart/form-data handling
3. Implement chunked file uploads for large files
4. Calculate file checksums during upload
5. Implement download with streaming
6. Create quota management system
7. Implement file deletion with cleanup
8. Add virus scanning integration point (optional)
9. Create storage monitoring utilities

Module 6: User Service & Management

Purpose: Implement user registration, profile management, and user-related operations.

Files to Create:

- `app/services/user_service.py` - User business logic
- `app/api/routes/users.py` - User endpoints
- `tests/test_users.py` - User management tests

Key Concepts to Learn:

- User registration and validation
- Email verification flow
- Password change functionality
- User profile management
- Account deactivation
- Pagination and filtering
- User search functionality

Tools & Libraries:

```
email-validator==2.1.0
sendgrid==6.10.0 (Optional for email verification)
```

Learning Resources:

- User registration best practices
- Email verification patterns
- Password change security
- User data privacy considerations

User Endpoints:

```
POST /api/v1/users/register - Create new user
POST /api/v1/users/login - User login
POST /api/v1/users/refresh - Refresh token
POST /api/v1/users/logout - User logout
GET /api/v1/users/me - Get current user
PUT /api/v1/users/{user_id} - Update user profile
POST /api/v1/users/{user_id}/change-password - Change password
DELETE /api/v1/users/{user_id} - Delete user account
GET /api/v1/users - List users (admin only)
```

Tasks to Complete:

1. Implement user registration with validation

2. Implement user login with token generation
3. Implement token refresh mechanism
4. Create profile update endpoint
5. Create password change endpoint
6. Implement soft delete for user accounts
7. Add email verification (optional)
8. Create admin user management endpoints
9. Write comprehensive user service tests

Module 7: Bucket Management

Purpose: Implement bucket operations (create, list, delete, configure access).

Files to Create:

- `app/services/bucket_service.py` - Bucket business logic
- `app/api/routes/buckets.py` - Bucket endpoints
- `tests/test_buckets.py` - Bucket operation tests

Key Concepts to Learn:

- Bucket creation and validation
- Ownership and access control
- Bucket naming rules
- Storage quota per bucket
- Bucket versioning (optional)
- Bucket lifecycle policies (optional)
- Public/private bucket settings

Learning Resources:

- S3 bucket naming conventions
- Access control patterns
- Resource ownership verification

Bucket Endpoints:

```
POST /api/v1/buckets - Create bucket
GET /api/v1/buckets - List user's buckets
GET /api/v1/buckets/{bucket_id} - Get bucket details
PUT /api/v1/buckets/{bucket_id} - Update bucket settings
DELETE /api/v1/buckets/{bucket_id} - Delete bucket
GET /api/v1/buckets/{bucket_id}/storage - Get bucket storage stats
```

Tasks to Complete:

1. Create bucket with ownership validation
2. Implement bucket naming rules
3. List buckets with pagination and filtering
4. Retrieve bucket details and statistics
5. Update bucket settings (quota, public/private)
6. Delete bucket with file cleanup
7. Implement bucket access control
8. Add storage quota enforcement
9. Write bucket service tests

Module 8: File Operations & API Endpoints

Purpose: Implement file upload, download, list, delete, and metadata operations.

Files to Create:

- `app/api/routes/files.py` - File endpoints
- `tests/test_files.py` - File operation tests

Key Concepts to Learn:

- Multipart file upload handling
- File streaming and downloads
- HTTP range requests for partial downloads
- File listing with pagination

- File metadata operations
- File deletion and cleanup
- CORS for file access
- Content-Type and Content-Disposition headers

Learning Resources:

- HTTP file upload best practices
- Streaming large files
- Range request implementation (HTTP 206)
- FastAPI file response documentation

File Endpoints:

```
POST /api/v1/buckets/{bucket_id}/files - Upload file
GET /api/v1/buckets/{bucket_id}/files - List files in bucket
GET /api/v1/buckets/{bucket_id}/files/{file_id} - Get file metadata
GET /api/v1/buckets/{bucket_id}/files/{file_id}/download - Download file
PUT /api/v1/buckets/{bucket_id}/files/{file_id} - Update file metadata
DELETE /api/v1/buckets/{bucket_id}/files/{file_id} - Delete file
```

Tasks to Complete:

1. Implement single file upload
2. Implement multipart/chunked file upload
3. Implement file listing with pagination and filters
4. Implement file download with streaming
5. Implement range request support
6. Create file metadata update endpoint
7. Implement file deletion with storage cleanup
8. Add file search and filtering capabilities
9. Implement file tagging/metadata (optional)
10. Write comprehensive file operation tests

Module 9: Access Control & Permissions

Purpose: Implement authorization and permission management for buckets and files.

Files to Create:

- `app/core/permissions.py` - Permission checking utilities
- `app/services/access_control_service.py` - Access control logic
- `app/api/routes/access_control.py` - Permission endpoints
- `tests/test_permissions.py` - Permission tests

Key Concepts to Learn:

- Role-based access control (RBAC)
- Resource-level permissions
- Permission inheritance
- Share-with functionality
- Public/private resource settings
- Admin capabilities
- Permission verification in endpoints

Permission Levels:

```
- owner: Full control
- admin: Manage access and delete
- write: Upload and modify files
- read: Download files only
```

Learning Resources:

- RBAC design patterns
- Permission matrix design
- Authorization best practices
- Zero-trust security model

Access Control Endpoints:

```
POST /api/v1/buckets/{bucket_id}/share - Grant bucket access
GET /api/v1/buckets/{bucket_id}/access - List bucket access
DELETE /api/v1/buckets/{bucket_id}/access/{user_id} - Revoke access
POST /api/v1/buckets/{bucket_id}/files/{file_id}/share - Share file
```

Tasks to Complete:

1. Create permission verification utility
2. Implement FastAPI dependency for permission checking
3. Create access control service
4. Implement bucket sharing functionality
5. Implement file-level permissions
6. Create permission update endpoints
7. Implement permission inheritance logic
8. Add audit logging for permission changes
9. Write comprehensive permission tests

Module 10: Background Processing & Tasks

Purpose: Implement background task processing for long-running operations.

Files to Create:

- `app/tasks/background_tasks.py` - FastAPI background tasks
- `app/tasks/celery_app.py` - Celery configuration (optional)
- `app/tasks/celery_tasks.py` - Celery task definitions
- `tests/test_tasks.py` - Task tests

Key Concepts to Learn:

- FastAPI BackgroundTasks for simple async operations
- Celery for distributed task processing
- Task queuing and retry logic
- Progress tracking for long operations

- Task scheduling (Celery Beat)
- Error handling in background tasks
- Monitoring task execution

Tools & Libraries:

```
celery==5.3.4 (Optional)
redis==5.0.1 (For Celery broker - optional)
```

Learning Resources:

- FastAPI background tasks documentation
- Celery documentation and best practices
- Task design patterns
- Error handling in async tasks

Background Tasks:

- Virus scanning on upload
- Image optimization and thumbnail generation
- Large file cleanup
- Storage quota notifications
- Report generation
- Batch operations

Tasks to Complete:

1. Implement file cleanup background task
2. Implement storage quota check task
3. Implement image thumbnail generation
4. Set up Celery (optional advanced setup)
5. Create task status tracking
6. Implement task retries with exponential backoff
7. Add task progress tracking for long operations
8. Create task monitoring utilities

9. Write task execution tests

Module 11: Error Handling & Logging

Purpose: Implement comprehensive error handling and application logging.

Files to Create:

- `app/core/exceptions.py` - Custom exception classes
- `app/core/logger.py` - Logging configuration
- `app/middleware/error_handlers.py` - Global error handlers
- `tests/test_error_handling.py` - Error handling tests

Key Concepts to Learn:

- Custom exception hierarchy
- Global exception handlers in FastAPI
- Structured logging
- Log levels and formatting
- Request/response logging
- Error tracking and reporting
- Correlation IDs for request tracing

Tools & Libraries:

```
python-json-logger==2.0.7
loguru==0.7.2 (Optional, for better logging)
sentry-sdk==1.38.0 (Optional, for error tracking)
```

Learning Resources:

- FastAPI exception handling documentation
- Structured logging best practices
- Error response standardization

Custom Exceptions:

```
- AuthenticationException
- AuthorizationException
- ValidationException
- ResourceNotFoundException
- QuotaExceededException
- StorageException
- FileAlreadyExistsException
```

Tasks to Complete:

1. Create custom exception hierarchy
2. Create global exception handlers
3. Implement structured logging configuration
4. Add request/response logging middleware
5. Implement correlation ID tracking
6. Create standardized error response format
7. Add Sentry integration (optional)
8. Write error handling tests
9. Create logging documentation

Module 12: Testing & Quality Assurance

Purpose: Implement comprehensive testing strategy and quality assurance.

Files to Create:

- `tests/conftest.py` - Test fixtures and configuration
- `tests/test_auth.py` - Authentication tests
- `tests/test_users.py` - User service tests
- `tests/test_buckets.py` - Bucket operation tests
- `tests/test_files.py` - File operation tests
- `tests/test_permissions.py` - Permission tests
- `tests/test_storage.py` - Storage operation tests
- `tests/test_integration.py` - Integration tests

- `.coveragerc` - Code coverage configuration
- `pytest.ini` - Pytest configuration

Key Concepts to Learn:

- Unit testing best practices
- Integration testing strategies
- Mocking and fixtures
- Test database setup
- Code coverage measurement
- Continuous integration
- Test organization and naming conventions
- API endpoint testing with TestClient

Tools & Libraries:

```
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
httpx==0.25.2
pytest-mock==3.12.0
factory-boy==3.3.0
faker==21.0.0
```

Learning Resources:

- Pytest documentation
- Fixture usage patterns
- Testing async code
- API testing best practices
- Code coverage targets

Test Coverage Areas:

- Authentication (login, token refresh, token validation)
- User registration and management

- Bucket creation and management
- File upload, download, delete
- Permission checking
- Error handling and exceptions
- Edge cases and boundary conditions

Tasks to Complete:

1. Set up pytest and test configuration
2. Create test fixtures for users, buckets, files
3. Write authentication tests
4. Write user management tests
5. Write bucket operation tests
6. Write file operation tests
7. Write permission verification tests
8. Write storage operation tests
9. Write integration tests for complex workflows
10. Achieve 80%+ code coverage
11. Set up CI/CD pipeline (GitHub Actions/GitLab CI)

Module 13: API Documentation & Standards

Purpose: Create comprehensive API documentation and follow REST standards.

Files to Create:

- `docs/API.md` - API documentation
- `docs/AUTHENTICATION.md` - Auth documentation
- `docs/EXAMPLES.md` - API usage examples
- `app/api/v1/__init__.py` - API versioning

Key Concepts to Learn:

- OpenAPI/Swagger specification

- RESTful API design principles
- API versioning strategies
- Request/response examples
- Error code documentation
- API rate limiting documentation
- Deprecation policies

Learning Resources:

- OpenAPI 3.0 specification
- REST API best practices
- API documentation standards
- Semantic versioning

API Standards:

```
Base URL: /api/v1
Response Format: JSON
Authentication: Bearer token in Authorization header
Error Response: {
  "detail": "error message",
  "error_code": "ERROR_CODE",
  "timestamp": "2024-01-09T10:00:00Z"
}
```

Tasks to Complete:

1. Document all API endpoints with examples
2. Create authentication guide
3. Create quickstart guide
4. Create error code reference
5. Generate OpenAPI schema
6. Set up API documentation UI (Swagger/ReDoc)
7. Create webhook documentation (if applicable)
8. Write migration guide for API versions

9. Create troubleshooting guide

Module 14: Deployment & DevOps

Purpose: Prepare application for production deployment.

Files to Create:

- `Dockerfile` - Docker container configuration
- `docker-compose.yml` - Multi-container orchestration
- `.dockerignore` - Docker build exclusions
- `nginx.conf` - Nginx reverse proxy configuration (optional)
- `.github/workflows/ci-cd.yml` - CI/CD pipeline
- `requirements-prod.txt` - Production dependencies
- `gunicorn.conf.py` - Gunicorn WSGI server configuration

Key Concepts to Learn:

- Docker containerization
- Docker Compose for local development
- Environment configuration management
- Application health checks
- Process managers (Gunicorn, Uvicorn)
- Reverse proxy setup (Nginx)
- SSL/TLS certificate management
- Secrets management
- Database backup and recovery
- Monitoring and alerting

Tools & Libraries:

```
gunicorn==21.2.0
docker (Docker installation)
docker-compose
```

Learning Resources:

- Docker official documentation
- FastAPI deployment guide
- Docker Compose documentation
- Nginx configuration guide
- GitHub Actions documentation

Deployment Checklist:

- Environment variables configured
- Database migrations applied
- Secret keys rotated
- Security headers configured
- CORS properly configured
- Rate limiting enabled
- Monitoring enabled
- Backup strategy in place

Tasks to Complete:

1. Create Dockerfile with multi-stage builds
2. Create docker-compose for local development
3. Set up environment-specific configurations
4. Configure Gunicorn/Uvicorn
5. Set up Nginx reverse proxy configuration
6. Implement health check endpoint
7. Set up CI/CD pipeline with GitHub Actions
8. Configure database backups
9. Create deployment documentation
10. Implement monitoring and alerting

Development Workflow

Phase 1: Foundation (Modules 1-4)

- Project setup and environment
- Database models and migrations
- Authentication and security
- Basic schemas and validation

Phase 2: Core Features (Modules 5-8)

- Storage and file management
- User service implementation
- Bucket management
- File operation endpoints

Phase 3: Advanced Features (Modules 9-10)

- Access control and permissions
- Background task processing
- Advanced file operations

Phase 4: Quality & Deployment (Modules 11-14)

- Error handling and logging
- Comprehensive testing
- API documentation
- Deployment and DevOps

Technology Stack Summary

Core:

- FastAPI 0.104+
- Python 3.11+

- PostgreSQL 14+ / SQLite 3.35+

Authentication & Security:

- python-jose (JWT)
- passlib + Argon2 (Password hashing)
- cryptography (Encryption)

Database:

- SQLAlchemy 2.0+ (ORM)
- Alembic (Migrations)
- psycopg2 (PostgreSQL driver)

File Handling:

- aiofiles (Async I/O)
- python-multipart (Form parsing)

Task Processing:

- FastAPI BackgroundTasks (built-in)
- Celery 5.3+ (Advanced)
- Redis (Celery broker)

Testing:

- pytest 7.4+
- httpx (HTTP client for tests)
- pytest-asyncio

Deployment:

- Docker & Docker Compose
- Gunicorn
- Nginx

Getting Started

1. **Clone and Setup:**

```bash
git clone <repo>
cd file-storage-api
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

2. **Configure Environment:**

```bash
cp .env.example .env
# Edit .env with your configuration
```

3. **Initialize Database:**

```bash
alembic upgrade head
```

4. **Run Application:**

```bash
uvicorn app.main:app --reload
```

5. **Access Documentation:**
   - Swagger UI: http://localhost:8000/docs
   - ReDoc: http://localhost:8000/redoc

Additional Resources

- **FastAPI Documentation:** https://fastapi.tiangolo.com

- **SQLAlchemy Documentation:** https://docs.sqlalchemy.org
- **Pydantic Documentation:** https://docs.pydantic.dev
- **Alembic Documentation:** https://alembic.sqlalchemy.org
- **JWT Best Practices:** https://tools.ietf.org/html/rfc7519
- **OWASP Security Guidelines:** https://owasp.org

Project Structure Template

```
file-storage-api/
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── core/
│   │   ├── __init__.py
│   │   ├── config.py
│   │   ├── database.py
│   │   ├── security.py
│   │   ├── exceptions.py
│   │   ├── logger.py
│   │   └── permissions.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── base.py
│   │   ├── user.py
│   │   ├── bucket.py
│   │   ├── file.py
│   │   └── access_control.py
│   ├── schemas/
│   │   ├── __init__.py
│   │   ├── user.py
│   │   ├── bucket.py
│   │   ├── file.py
│   │   └── common.py
│   ├── services/
│   │   ├── __init__.py
│   │   ├── auth_service.py
│   │   ├── user_service.py
│   │   ├── bucket_service.py
│   │   ├── file_service.py
```

```
│   │   ├── storage_service.py
│   │   └── access_control_service.py
│   ├── api/
│   │   ├── __init__.py
│   │   ├── dependencies.py
│   │   └── v1/
│   │       ├── __init__.py
│   │       ├── endpoints/
│   │       │   ├── __init__.py
│   │       │   ├── auth.py
│   │       │   ├── users.py
│   │       │   ├── buckets.py
│   │       │   ├── files.py
│   │       │   └── access_control.py
│   │       └── router.py
│   ├── middleware/
│   │   ├── __init__.py
│   │   └── error_handlers.py
│   └── tasks
```