AmlgoLabs Backend Detailed Documentation

1) AmlgoLabs Backend (API) Flow and Logic

Architecture Overview

The AmlgoLabs backend infrastructure is built on Next.js API routes that create a cohesive RESTful API system. The architecture implements a clean separation of concerns through modular components:

- **API Routes Layer**: Handles incoming HTTP requests, processes data, and returns appropriate responses
- **Data Layer**: Manages database interactions through Mongoose models
- **Validation Layer**: Ensures data integrity before processing
- **Service Layer**: Handles business logic including email notifications
- **Error Handling Layer**: Provides consistent error management across all endpoints

Core API Endpoints

Contact Form Submission Endpoint (`contact-us.js`)

Detailed Flow:

1. **Request Reception**: The endpoint receives a POST request containing contact form data
2. **Database Connection**: Establishes connection to MongoDB using connection pooling
3. **Data Extraction**: Parses JSON body to extract name, email, message, and phone fields
4. **Validation Process**:
    - Applies Joi validation schema to ensure all required fields meet criteria
    - Name must be at least 2 characters
    - Email must be valid format
    - Message must be at least 5 characters
    - Phone must match pattern if provided
5. **Database Operation**:
    - Creates new ContactUs document instance
    - Saves record with timestamp to MongoDB
6. **Email Notification Process**:
    - Constructs email parameters using template

    - Configures AWS SES client with proper credentials
    - Sends notification email to admin with all contact details
    - Email includes sender details for easy response
7. **Response Generation**:
    - Returns 201 status with success message on completion
    - Returns 400 status with validation errors if validation fails

Error Handling:

- Validation errors return specific feedback on which fields failed validation
- Server errors are caught by asyncHandler and return appropriate 500 status code
- All errors are logged to console for monitoring

Job Application Submission Endpoint (`job-application.js`)

Detailed Flow:

1. **Request Reception**: Receives multipart form data POST request containing applicant information and resume file
2. **Database Connection**: Establishes MongoDB connection
3. **Form Data Processing**:
    - Extracts fields: name, email, phone, coverLetter, CV file, jobId, and jobTitle
    - Special handling for binary file data from form
4. **Validation Process**:
    - Validates all text fields using Joi schema
    - Performs custom file validation:
        - Verifies file is present
        - Validates file type (PDF, DOC, DOCX only)
        - Ensures file size does not exceed 5MB limit
5. **File Storage Process**:
    - Generates unique filename with timestamp and random string
    - Creates upload directory if not exists
    - Writes file buffer to server filesystem
    - Constructs accessible URL path to the resume
6. **Database Operation**:

- Creates new JobApplication document with all fields and resume URL
- Saves record with timestamp
- Retrieves generated MongoDB ID for later reference

7. **Email Notification Process**:
   - Sends notification email to company HR with all application details
   - Includes secure link to view/download resume
   - Sends confirmation email to applicant with job title reference
   - CC's HR department on confirmation email
8. **Response Generation**:
   - Returns 201 status with success message
   - Returns 400 status with validation errors if validation fails

Error Handling:

- Comprehensive validation error reporting for all fields including file-specific errors
- Server-side errors return 500 status with error details
- All errors logged for monitoring purposes

Resume Viewer Endpoint (`view-resume/[id]/route.js`)

Detailed Flow:

1. **Request Reception**: Receives GET request with application ID parameter
2. **Database Connection**: Connects to MongoDB
3. **Document Retrieval**:
   - Queries JobApplication collection by ID
   - Validates that application and resume URL exist
4. **File Processing**:
   - Extracts filename from resume URL
   - Constructs absolute file path on server
   - Reads file into memory buffer
5. **Response Generation**:
   - Determines MIME type based on file extension (.pdf, .docx, .doc)
   - Sets appropriate content-type headers
   - Creates sanitized download filename from applicant name

- Sets content-disposition header (inline for PDFs, attachment for other types)
- Returns file buffer as response body

6. **Error Handling**:
    - Returns 404 if application not found
    - Returns 500 for any server-side errors
    - Logs detailed error information to console

Utility Functions and Services

Email System Architecture

Template Management:

- Email templates are defined as JavaScript objects with structured format
- Each template contains:
    - Source email address
    - Subject line (static or dynamic function)
    - Body content (plain text or HTML)
- Templates are processed through utility functions that inject dynamic content

Email Templates:

1. **contactDetailsToAdmin**:
    - Purpose: Notifies admin team of new contact form submission
    - Format: Plain text with formatted contact details
    - Dynamic fields: Name, email, phone, message content
2. **contactUsThanks**:
    - Purpose: Confirms receipt of contact form to user
    - Format: Plain text acknowledgment
    - Dynamic fields: Name
3. **jobApplicationToAmlgoLabs**:
    - Purpose: Notifies HR of new job application
    - Format: Plain text with application details
    - Dynamic fields: Name, email, phone, resume URL, cover letter, job ID, job title
4. **jobApplicationThanks**:
    - Purpose: Confirms receipt of job application to applicant

    - Format: HTML formatted acknowledgment
    - Dynamic fields: Name, job title

AWS SES Integration:

- Utilizes AWS SDK for JavaScript v3
- Configures SES client with AWS credentials from environment variables
- Implements SendEmailCommand for reliable email delivery
- Supports CC addresses for HR department notifications
- Handles both plain text and HTML email formats

Error Handling System

asyncHandler Utility:

- Purpose: Standardizes error handling across all API endpoints
- Implementation:
    - Wraps API handler functions for consistent error management
    - Converts function results to proper Response objects
    - Catches and formats any thrown exceptions
    - Ensures consistent JSON response format
    - Automatically adds appropriate content-type headers
    - Logs all errors for debugging purposes

Validation System

Contact Form Validation:

- Uses Joi schema validation library
- Validates:
    - Name: Required, minimum 2 characters
    - Email: Required, valid email format
    - Message: Required, minimum 5 characters
    - Phone: Optional, matches international or local format pattern
- Provides custom error messages for each validation rule

Job Application Validation:

- Validates all text fields with standard rules
- Custom validation for resume file:
  - Verifies instance of File object
  - Validates MIME type against whitelist
  - Enforces 5MB size limit
- Provides detailed error messages for each field

2) Database

Connection Architecture

The database connection implements a singleton pattern to optimize performance:

- Uses Mongoose ODM (Object Document Mapper) for MongoDB interaction
- Maintains persistent connection throughout application lifecycle
- Implements connection state checking to prevent redundant connections
- Configures MongoDB with optimized connection parameters:
  - `useNewUrlParser`: Ensures compatibility with MongoDB connection string format
  - `useUnifiedTopology`: Enables modern MongoDB connection management

Data Models

ContactUs Model

Schema Definition:

- `name`: String
  - Required field
  - Stores the full name of the contact
  - Default empty string if not properly validated
- `email`: String
  - Required field
  - Contains validated email address
  - Used for communication with the contact
  - Default empty string if not properly validated
- `message`: String
  - Required field

    - Contains the full message body
    - No maximum length restriction
    - Default empty string if not properly validated
- `phone`: String
    - Optional field
    - Stores formatted phone number
    - No specific format enforcement at database level
    - Empty string default if not provided
- `createdAt`: Date
    - Automatically generated timestamp
    - Records when the contact form was submitted
    - Used for sorting and reporting

Purpose: Stores all contact form submissions for customer relationship management and followup

JobApplication Model

Schema Definition:

- `name`: String
    - Required field
    - Stores applicant's full name
    - Used for identification and communication
- `email`: String
    - Required field
    - Contains validated email address
    - Used for application status updates
- `phone`: String
    - Required field
    - Stores applicant's contact number
    - Used for interview scheduling
- `coverLetter`: String
    - Optional field
    - Stores cover letter text

    - May be empty string if not provided
- `resumeUrl`: String
    - Required field
    - Contains path to uploaded resume file
    - Format: `/uploads/[unique-filename].[extension]`
- `createdAt`: Date
    - Automatically generated timestamp
    - Records when application was submitted
    - Used for sorting and reporting

Purpose: Stores all job applications with references to uploaded resume files for the hiring process

Database Operations

- **Read Operations**: Used in resume retrieval by ID
- **Write Operations**: Used when saving new contact forms and job applications
- **No Update or Delete Operations**: The current implementation does not modify or remove existing records

3) Deployment

Server Environment

- **Platform**: Hostinger Linux Server
- **Operating System**: Linux (specific distribution not specified)
- **Web Server**: Nginx
- **SSL Provider**: Let's Encrypt via Certbot
- **Process Manager**: PM2

Detailed Deployment Process

1. Code Deployment

```bash
# Clone repository from Git
git clone [repository-url] /path/to/app
cd /path/to/app

# Install dependencies
npm install

# Create and configure environment file
cp .env.example .env.local
nano .env.local  # Edit environment variables

# Build application for production
npm run build
```

2. Environment Configuration

The `.env.local` file must be configured with the following variables:

- `AWS_ACCESS_KEY_ID`: AWS credential for SES email service
- `AWS_SECRET_ACCESS_KEY`: AWS credential for SES email service
- `NEXT_PUBLIC_BASE_URL_PROD`: Production URL (e.g., https://www.amlgolabs.com)
- `NEXT_PUBLIC_BASE_URL_DEV`: Development URL (for testing)
- `MONGODB_URI`: Full MongoDB connection string including credentials and database name
- `NODE_ENV`: Set to "production" for production deployment

3. PM2 Process Management

```bash
# Start application with PM2
pm2 start npm --name "amlgolabs" -- start

# Ensure PM2 restarts application on server reboot
pm2 startup
pm2 save

# Monitor application status
pm2 status
pm2 logs amlgolabs
```

4. Nginx Configuration

```bash
# Create Nginx configuration file
sudo nano /etc/nginx/sites-available/amlgolabs.com

# Create symbolic link to enable site
sudo ln -s /etc/nginx/sites-available/amlgolabs.com /etc/nginx/sites-enabled/

# Test Nginx configuration
sudo nginx -t

# Restart Nginx to apply changes
sudo systemctl restart nginx
```

Sample Nginx configuration:

```
server {
    listen 80;
    server_name amlgolabs.com www.amlgolabs.com;

    location / {
        proxy_pass http://localhost:3000;  # Next.js default port
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    # Configure larger upload size for resume uploads
    client_max_body_size 10M;
}
```

5. SSL Certificate Installation

bash

```bash
# Install Certbot
sudo apt install certbot python3-certbot-nginx

# Obtain and install SSL certificate
sudo certbot --nginx -d amlgolabs.com -d www.amlgolabs.com

# Verify auto-renewal is configured
sudo systemctl status certbot.timer
```

6. File System Permissions

bash

```bash
# Create uploads directory with proper permissions
mkdir -p /path/to/app/public/uploads
chmod 755 /path/to/app/public/uploads

# Set ownership to web server user
chown -R www-data:www-data /path/to/app/public/uploads
```

Maintenance Procedures

Log Management

```bash
# View application logs
pm2 logs amlgolabs

# View Nginx access and error logs
sudo tail -f /var/log/nginx/access.log
sudo tail -f /var/log/nginx/error.log
```

Updates and Redeployment

```bash
# Pull latest code
cd /path/to/app
git pull

# Install any new dependencies
npm install

# Rebuild application
npm run build

# Restart application
pm2 restart amlgolabs
```

SSL Certificate Renewal

SSL certificates from Let's Encrypt automatically renew via Certbot's timer service. Manual renewal if needed:

```bash
sudo certbot renew
```

Security Considerations

- **File Upload Security**:
  - File type validation restricts to PDF, DOC, DOCX only

  - Size limitation prevents DoS attacks via large files
  - Unique filenames prevent overwriting existing files
  - Files served through authenticated API endpoint only
- **Input Validation**:
  - All user inputs validated before processing
  - Custom validation rules prevent injection attacks
  - Error messages designed to avoid information leakage
- **API Security**:
  - All communications encrypted via SSL/TLS
  - Sensitive credentials stored in environment variables
  - No direct database access exposed to clients
- **Network Security**:
  - Nginx configured as reverse proxy
  - Only necessary ports exposed
  - SSL certificates automatically renewed