Resume System – Dynamic Resume Builder and Verification Platform

Build modern resumes programmatically. This backend provides secure authentication, CRUD for projects/achievements/skills, automatic resume maintenance, verification of items, and insights—all powered by Node.js, Express, and MongoDB.

> ⚠️ **Note:** All API endpoints expect request bodies in **JSON format**.
> Use Postman (or any API client) → Body → `raw` → `JSON` when testing.

Key Features

- **JWT auth**: Register/Login with hashed passwords and stateless tokens
- **Modular CRUD**: Projects, Achievements, Skills with verification flags
- **Auto-maintained Resume**: `Resume` embeds arrays of references and updates on every change
- **AI Summary Generator**: OpenRouter-powered professional summary generation based on user data
- **PDF Resume Export**: Generate professional, ATS-friendly resume PDFs with dynamic content
- **External Platform Integration**: Simulated system for auto-updating resumes from external learning platforms
- **Analytics/Insights**: Quick stats of verified/unverified items per user
- **Resource Optimization**: Prevents server waste by validating data before processing
- **Robust DX**: `ApiError`, `ApiResponse`, and `asyncHandler` for clean APIs
- **AI Summary Generator**: OpenRouter-powered professional summary generation based on user data.
  > 🔗 **Detailed Guide** → See OpenRouter DeepSeek Integration for full instructions, outputs, and example requests.

Table of Contents

📘 Related Documentation

- AI Summary Generation Guide: Detailed explanation of how AI-based summary generation works, request/response structures, and common pitfalls.
- OpenRouter DeepSeek Integration: Step-by-step guide to integrating OpenRouter DeepSeek V3.1 for AI-powered resume summaries, including example outputs and request formats.
- System Architecture & Workflow: Visual explanation of backend architecture, data flow, and system components.
- External Platform Integration: Complete guide to the external platform integration simulation system that auto-updates resumes from learning platforms.

Introduction

Problem: Manually maintaining resumes across projects, courses, and achievements is error-prone and time-consuming.

Solution: A backend that lets users securely manage portfolio data. As users add items, their `Resume` is updated in real time. Admin/verifiers can mark items as verified. Consumers can fetch the full resume or high-level stats.

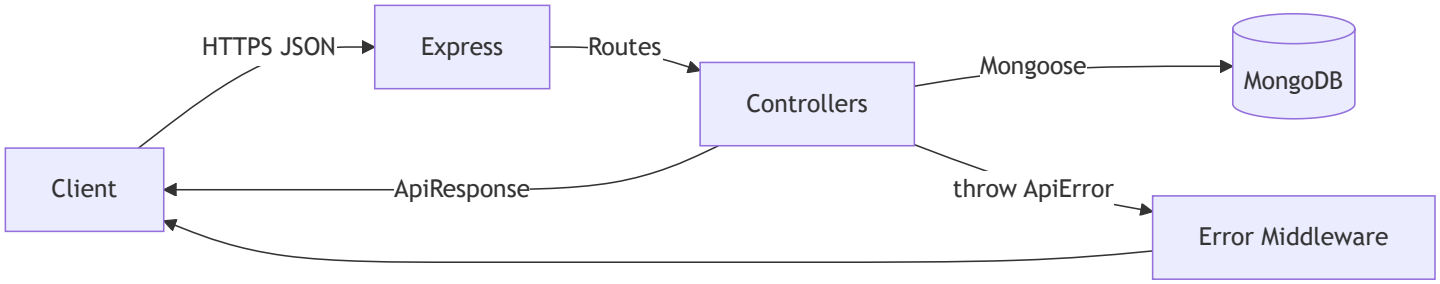
Feature Overview

- Auth: Register, Login (JWT); role support (`student` , `admin`)
- Resume: Summary, education, experience, and references to projects/achievements/skills
- Projects/Achievements/Skills: CRUD with `verified` flags
- Verification: Role-protected endpoint to mark items verified
- Insights: Aggregated counts (verified/unverified)
- AI Summary: OpenRouter-powered professional summary generation
- PDF Export: Professional resume PDF generation with dynamic content

- External Platform Integration: Simulated system for auto-updating resumes from external learning platforms (Internshala, Coursera, Devpost, etc.)

System Architecture

High-level flow: client → REST API → controllers → services/model ops → MongoDB. Error handling is centralized.



Tech stack:

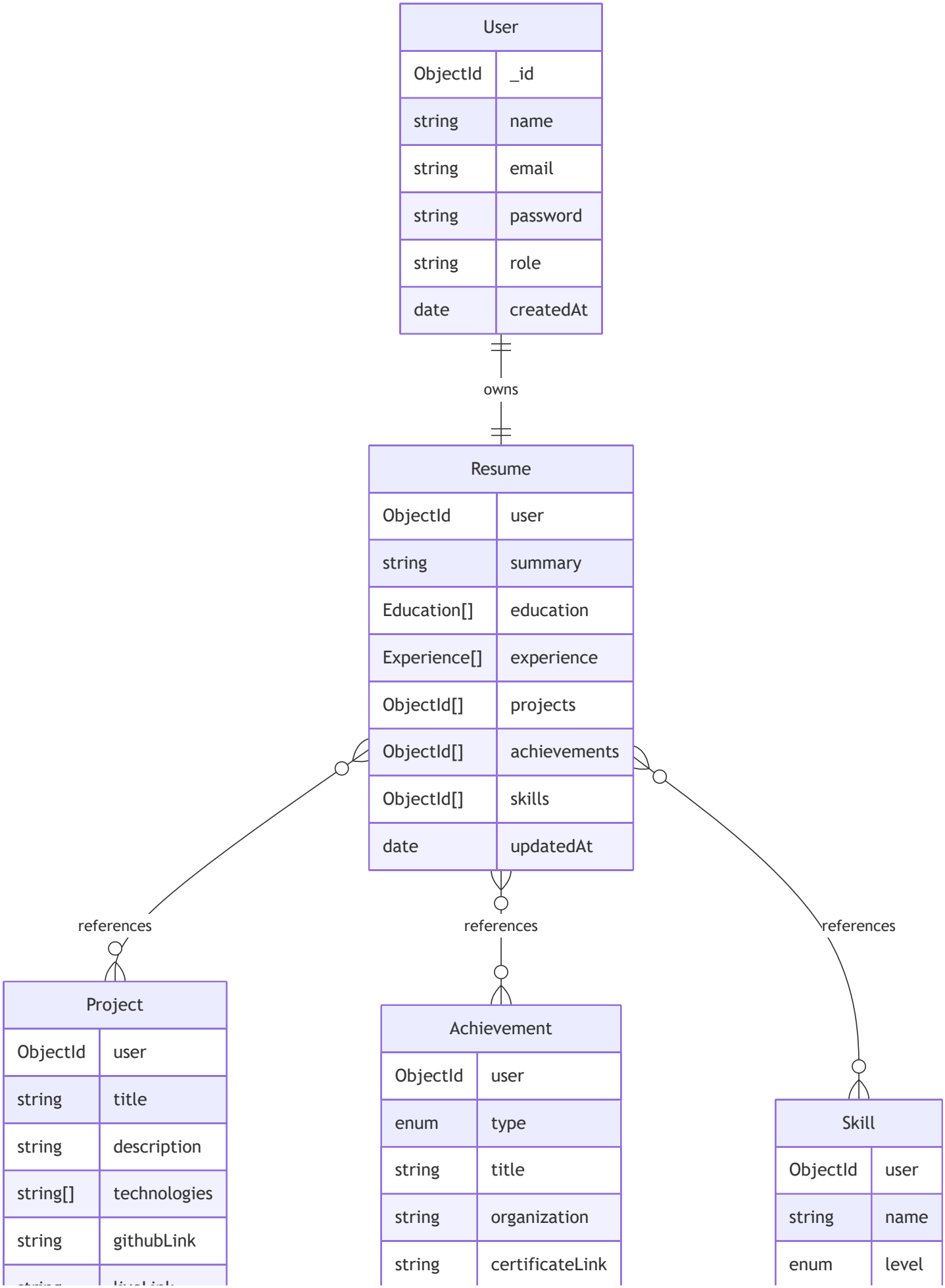
- Node.js, Express.js
- MongoDB with Mongoose
- JWT for auth
- Utilities: `ApiError`, `ApiResponse`, `asyncHandler`

Folder structure (key parts):

```
DB/
middlewares/
models/
controllers/
routes/
utils/
index.js
```

Models and Database Schema

Relationships: `User` 1—1 `Resume`; `Resume` holds refs to `Project`, `Achievement`, and `Skill` arrays.


User
ObjectId _id
string name
string email
string password
string role
date createdAt
owns
Resume
ObjectId user
string summary
Education[] education
Experience[] experience
ObjectId[] projects
ObjectId[] achievements
ObjectId[] skills
date updatedAt
references
Project
ObjectId user
string title
string description
string[] technologies
string githubLink
references
Achievement
ObjectId user
enum type
string title
string organization
string certificateLink
references
Skill
ObjectId user
string name
enum level

string	liveLink
bool	verified
date	createdAt

bool	verified
date	date

API Endpoints

Base URL: `/api`

> 🎯 **API Design Note**: All endpoints follow RESTful conventions with consistent HTTP methods and resource-based URIs. The same resource (e.g., `/achievements`) is accessed via different HTTP methods (GET, POST, PUT, DELETE) rather than using confusing endpoint names like `get-achievement` or `update-achievement` . This makes the API intuitive and follows industry standards.

Authentication

- POST `/auth/register`
  - Headers: `Content-Type: application/json`
  - Body:

    ```
    { "name": "Asjad Dawre", "email": "asjad@example.com", "password": "secret123" }
    ```

  - 201 Response:

    ```
    { "statusCode": 201, "data": { /* user */ }, "message": "User registered successfully", "succ
    ```

- POST `/auth/login`
  - Body:

    ```
    { "email": "asjad@example.com", "password": "secret123" }
    ```

  - 200 Response:

    ```
    { "statusCode": 200, "data": { "token": "<JWT>", "user": { /* user */ } }, "message": "Login
    ```

Projects (cookie-based auth; httpOnly cookie `accessToken` is set by login)

- POST `/projects`
  - Body:

    ```
    { "title": "Portfolio", "description": "Site", "technologies": ["React","Node"], "githubLink"
    ```

  - 201 Response: project created and resume updated
- GET `/projects`

    - 200 Response: list of user projects
- PUT `/projects/:projectId`
    - Body: partial update (e.g., `{ "verified": true }` by verifier)
    - 200 Response: updated project and resume `updatedAt`
- DELETE `/projects/:projectId`
    - 200 Response: deleted and removed from resume

Achievements (requires auth)

- POST `/achievements`
- GET `/achievements`
- PUT `/achievements/:achievementId`
- DELETE `/achievements/:achievementId`

Skills (requires auth)

- POST `/skills`
- GET `/skills`
- PUT `/skills/:skillId`
- DELETE `/skills/:skillId`

Resume (requires auth)

- GET `/resume` — returns populated resume (projects, achievements, skills)
- PUT `/resume/summary` — `{ "summary": "..." }` (manual update)
- POST `/resume/generate-summary` — AI-generated summary based on user data
- POST `/resume/preview-summary` — preview AI summary without saving
- PUT `/resume/education` — `{ "education": [ {"institute":"...","degree":"...","startYear":2022,"endYear":2026} ] }`
- PUT `/resume/experience` — `{ "experience": [ {"title":"SWE Intern","company":"X","duration":"Jun-Sep","description":"..."} ] }`
- GET `/resume/stats` — insights of verified/unverified counts
- GET `/resume/pdf` — generate and download professional resume PDF

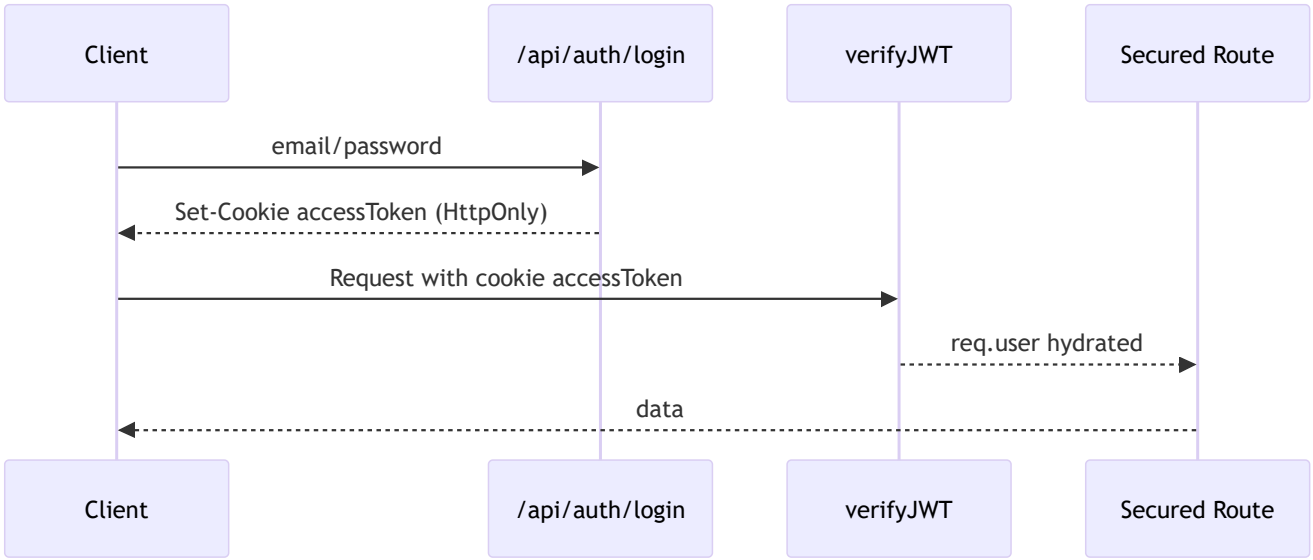
Verification (requires auth; typically elevated role)

- PUT `/verify/:type/:id` — `type` in `[project, achievement, skill, course]`
    - 200 Response: item marked `verified: true`

Authentication Flow

1. On login, server validates credentials and sets an httpOnly cookie `accessToken` (7d expiry).

2. Client does not manually handle the token; the browser sends the cookie automatically (with credentials enabled).
3. `verifyJWT` middleware reads the token from cookies and injects `req.user` for protected routes.

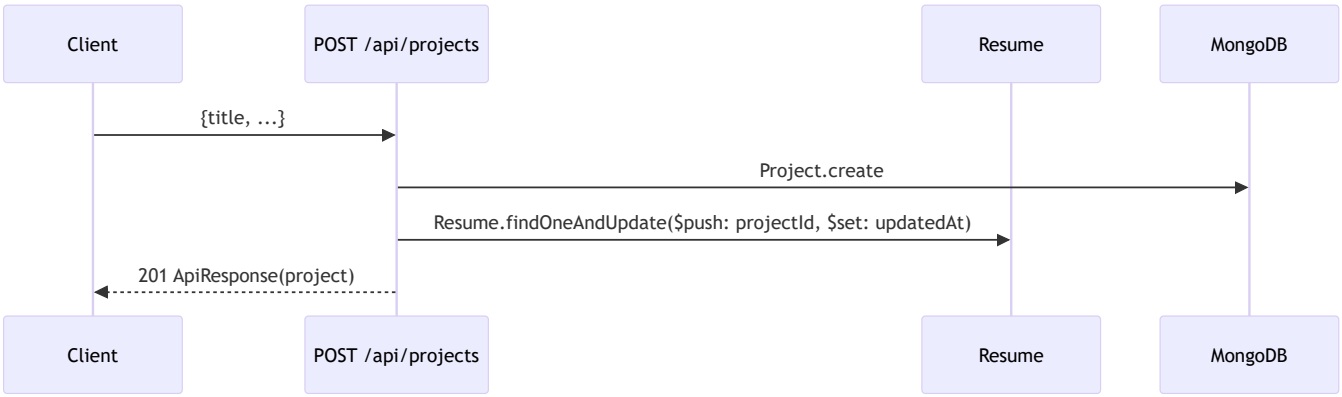


Resume Update Logic

When creating/updating/deleting a Project/Achievement/Skill:

- Create/Update/Delete the item document
- Update `Resume` accordingly (`$push` / `$pull` , set `updatedAt`)
- For summary/education/experience: the resume document is updated in place

Example lifecycle (create project):



Error and Response Handling

- Use `asyncHandler` to catch rejections and forward to the error middleware
- Throw `ApiError(message, statusCode)` inside controllers
- Always respond with `new ApiResponse(statusCode, data, message)` on success

Example error shape:

```
{ "statusCode": 400, "data": null, "message": "Project title is required", "success": false }
```

Verification System

Endpoint: `PUT /api/verify/:type/:id`

- Valid `type` : `project` | `achievement` | `skill` | `course`
- Sets `verified = true` on the target document
- Can be guarded by role checks (extend `verifyJWT` or add role middleware)

Resume Insights Endpoint

Endpoint: `GET /api/resume/stats`

- Computes counts from user's items
- Response example:

```
{
  "statusCode": 200,
  "data": {
    "totalProjects": 3,
    "verifiedProjects": 1,
    "totalAchievements": 2,
    "verifiedAchievements": 2,
    "totalSkills": 5,
    "verifiedSkills": 3
  },
  "message": "Resume stats fetched successfully",
  "success": true
}
```

PDF Resume Export

> 📸 **Image Quality Note**: If images appear unclear or low quality in the README, please view them directly in the `docs/test_output/` folder for better resolution.

The system includes a professional PDF resume generator that creates ATS-friendly, industry-standard resume PDFs with dynamic content based on user data.

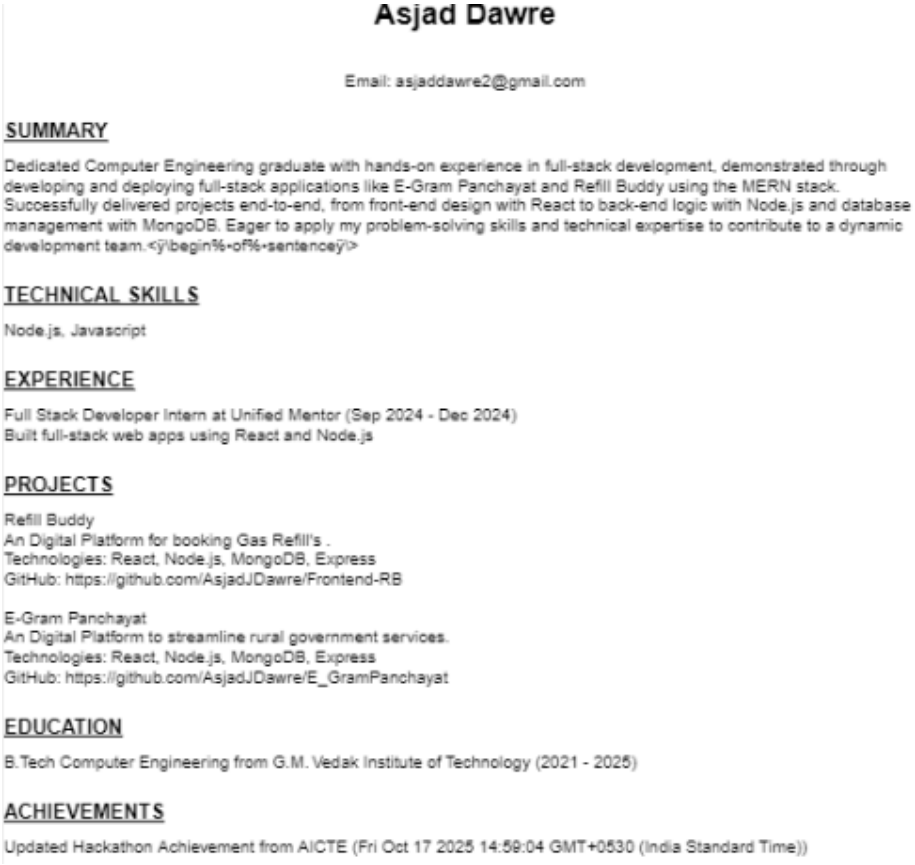
Endpoint: `GET /api/resume/pdf`

Features:

- **Professional Layout**: Clean, ATS-friendly design with proper typography
- **Dynamic Content**: Automatically populates with user's resume data
- **Section Order**: SUMMARY → TECHNICAL SKILLS → EXPERIENCE → PROJECTS → EDUCATION → ACHIEVEMENTS
- **Smart Project Selection**: Shows top 2 projects (prioritizes verified, then by date)
- **Graceful Data Handling**: Shows "Not Available" for empty sections, skips optional sections entirely
- **Resource Optimization**: Validates sufficient data before processing to prevent server waste

Sample Generated PDF:

> 📸 **Image Quality Note**: If images appear unclear or low quality in the README, please view them directly in the `docs/test_output/` folder for better resolution.



Asjad Dawre

Email: asjaddawre2@gmail.com

SUMMARY

Dedicated Computer Engineering graduate with hands-on experience in full-stack development, demonstrated through developing and deploying full-stack applications like E-Gram Panchayat and Refill Buddy using the MERN stack. Successfully delivered projects end-to-end, from front-end design with React to back-end logic with Node.js and database management with MongoDB. Eager to apply my problem-solving skills and technical expertise to contribute to a dynamic development team.<ÿ\begin%+of%+sentenceÿ>

TECHNICAL SKILLS

Node.js, Javascript

EXPERIENCE

Full Stack Developer Intern at Unified Mentor (Sep 2024 - Dec 2024)
Built full-stack web apps using React and Node.js

PROJECTS

Refill Buddy
An Digital Platform for booking Gas Refill's .
Technologies: React, Node.js, MongoDB, Express
GitHub: https://github.com/AsjadJDawre/Frontend-RB

E-Gram Panchayat
An Digital Platform to streamline rural government services.
Technologies: React, Node.js, MongoDB, Express
GitHub: https://github.com/AsjadJDawre/E_GramPanchayat

EDUCATION

B.Tech Computer Engineering from G.M. Vedak Institute of Technology (2021 - 2025)

ACHIEVEMENTS

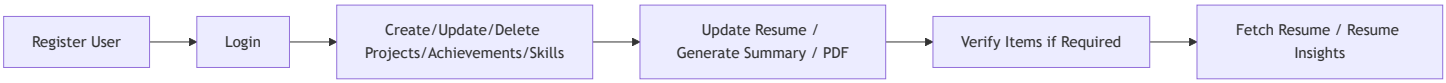
Updated Hackathon Achievement from AICTE (Fri Oct 17 2025 14:59:04 GMT+0530 (India Standard Time))

Usage-Example**:

How to Use the System

The recommended flow to use the External Platform Integration System is as follows:

1. **Register a User**
   Create a new account via `/auth/register`.
2. **Login**
   Authenticate using `/auth/login` to obtain an httpOnly cookie or JWT for subsequent requests.
3. **Use API Endpoints**
   - Create, update, delete **Projects**, **Achievements**, and **Skills**.
   - Update and fetch **Resume** (`/resume` endpoints).
   - Verify items using `/verify/:type/:id`.
   - Generate **AI Summary** and **PDF Resume**.
4. **Test the Flow**
   Use the Postman collection provided for a reproducible testing experience.



Postman Collection for Testing

A ready-to-use Postman collection is provided in `postman-collection/`.

> ⚠️ **Recommended Testing:** Use this collection to test all endpoints and workflows of the External Platform Integration System. It contains **pre-filled request bodies, URLs, and headers** that match the sample data in this documentation. This ensures results are reproducible and closely align with the developer's environment.

Usage:

1. Open Postman.
2. Import the collection from the `postman-collection/` folder.
3. Run requests in the order provided to simulate real data flows.
4. Verify responses against the outputs in the **Sample Output** section.

Response Headers:

- `Content-Type: application/pdf`
- `Content-Disposition: attachment; filename="resume_User_Name.pdf"`

Error Handling:

- **Insufficient Data**: Returns helpful message with guidance on required fields
- **Authentication**: Requires valid JWT token from httpOnly cookie
- **Security**: Users can only generate their own resume PDF

Insufficient Data Response:

```
{
    "statusCode": 400,
    "data": {
        "message": "Insufficient data for PDF generation",
        "suggestion": "Please add at least one of the following: projects, skills, achievements,
education, or experience before generating your resume PDF.",
        "requiredFields": [
            "projects",
            "skills",
            "achievements",
            "education",
            "experience"
        ]
    },
    "message": "Cannot generate PDF - insufficient resume data",
    "success": false
}
```

Future Scope

- Enhanced AI-generated resume summary with multiple templates
- **Live External Platform Integrations**: Real API connections to Udemy, Coursera, Hackathons, GitHub activity (currently simulated)
- PDF template customization and multiple layouts
- Advanced analytics and insights dashboard

> 🔗 **External Platform Integration**: Our system includes a comprehensive simulation of how external learning platforms (Internshala, Coursera, Devpost, HackerEarth, etc.) would automatically update user resumes. This simulation demonstrates the complete architecture for real-time resume updates based on external platform activities. **View Complete Documentation →**

Setup and Run

Prerequisites

- Node.js 18+

- MongoDB instance

Install

```
npm install
```

Environment

Create a `.env` file:

```
PORT=5000
DB_URL=mongodb://localhost:27017
DB_Name=resumesys
JWT_SECRET=supersecretkey
OPENROUTER_API_KEY=your_openrouter_api_key_here
```

Run

```
node index.js
```

Testing with Postman / curl (cookies)

- Set base URL to `http://localhost:5000/api`
- Ensure you send/receive cookies:
  - Postman: Enable "Send cookies" and turn on "Enable cookie management"; also set "Follow original HTTP method" for redirects if prompted. For CORS, your frontend client should use `withCredentials`.
  - curl: store cookies on login and resend them:

    ```
    # Login (store cookies)
    curl -i -c cookiejar.txt -X POST http://localhost:5000/api/auth/login \
      -H "Content-Type: application/json" \
      -d '{"email":"tester@example.com","password":"secret123"}'

    # Authenticated request (sends cookie)
    curl -s -b cookiejar.txt http://localhost:5000/api/projects

    # Generate and download resume PDF
    curl -s -b cookiejar.txt -X GET http://localhost:5000/api/resume/pdf \
      -o "my-resume.pdf"
    ```

Docker (Local) – Backend + MongoDB

Files included:

- `Dockerfile` – Node.js LTS base, installs deps, runs `npm run dev`
- `docker-compose.yml` – services: `backend`, `mongo` with volume and healthcheck
- `.dockerignore` – excludes `node_modules`, logs, and env files

Create `.env` for Docker/Local:

```
PORT=5000
DB_URL=mongodb://mongo:27017
DB_Name=ResumeSys
JWT_SECRET=supersecretkey
```

Build and run:

```
docker-compose build
docker-compose up
```

Access:

- Backend: `http://localhost:5000/` (health) and `http://localhost:5000/api`
- Mongo shell (optional):
  - Install mongosh locally, then: `mongosh mongodb://localhost:27017/ResumeSys`

Switching to MongoDB Atlas:

- Update `.env`:
  - `DB_URL=mongodb+srv://<user>:<pass>@<cluster>.mongodb.net`
  - `DB_Name=ResumeSys`
- Leave ports/compose unchanged; backend will connect to Atlas instead of the local `mongo` service.

Key commands and expected outputs

```bash
# 1) Build images
docker compose build
# Expected: build completes without error, shows "Successfully tagged resumesys-backend" (or similar)

# 2) Start in background
docker compose up -d
# Expected: containers created: resumesys-backend, resumesys-mongo (status: Up)

# 3) Check containers
docker ps
# Expected lines:
# resumesys-backend   0.0.0.0:5000->5000/tcp   Up
# resumesys-mongo     0.0.0.0:27017->27017/tcp Up

# 4) Tail backend logs
docker logs -f resumesys-backend
# Expected lines include:
# App listening at 5000
# Connected to ResumeSys

# 5) Probe health route
curl http://localhost:5000/
# Expected: "Resume System Backend Running..." or "Helllo from backend"

# 6) Register and Login (cookies)
curl -s -X POST http://localhost:5000/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"name":"Tester","email":"tester@example.com","password":"secret123"}'
# Expected: statusCode 201 with user object

curl -i -X POST http://localhost:5000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"tester@example.com","password":"secret123"}' \
  -c cookiejar.txt
# Expected: 200 with Set-Cookie: accessToken=...

# 7) Use cookie to access protected route
curl -s -b cookiejar.txt http://localhost:5000/api/projects
# Expected: statusCode 200 and an empty array [] initially (wrapped in ApiResponse)

# 8) Mongo shell (requires mongosh)
mongosh "mongodb://localhost:27017/ResumeSys" --eval "db.getCollectionNames()"
# Expected: collections like [ "users", "resumes", "projects", "achievements", "skills" ] after usage
```

Troubleshooting

- If the backend connects to Atlas instead of Docker Mongo, ensure `.env` has `DB_URL=mongodb://mongo:27017` and `DB_Name=ResumeSys` before `docker compose up`.
- If ports are busy, change `5000:5000` or `27017:27017` in `docker-compose.yml`.
- Restart sequence: `docker compose down && docker compose up -d`.

Conclusion

This backend centers on clarity, safety, and maintainability. It is well-suited for student portfolios, campus platforms, or any system that needs dynamic resume data with verification and insights.