INTERNSHIP & PLACEMENT

APPLICATION TRACKER

MVP Build Plan — Final Year B.Tech Project

Node.js + Express | MySQL | JWT Auth | REST API | 4-Week Plan

Solo / 2-Person Team	4 Weeks Timeline	Resume-Ready Project

IMPORTANT: This is a personal productivity tool, not a healthcare app. You own all the data, there is no regulatory compliance required, and the full stack is well within a beginner-to-intermediate skill level. Your goal is to show depth, not width.

1 Big Picture Vision — The Problem & Why It Matters

1.1 The Real Problem

Every B.Tech student applies to 30–100+ companies during placement season. You're copying rows into a spreadsheet, losing track of which company asked for a follow-up, forgetting whether you cleared the OA round for Infosys, and missing the deadline to reply to an offer letter. A plain spreadsheet has no workflow, no analytics, and no memory.

Your project solves this with a structured web application where every application has a lifecycle, notes are first-class citizens, and a dashboard tells you at a glance exactly how your placement season is going.

1.2 Why This Project Is Resume Gold

- Demonstrates end-to-end full-stack thinking: schema design → API → auth → business logic → analytics
- Uses industry-standard tools (Node.js, Express, MySQL, JWT) that every interviewer recognizes
- You can demo it live in any interview using a deployed URL — not a local screenshot
- The feature set naturally produces interview talking points: JWT security, relational joins, query optimization, CSV generation
- It is genuinely useful to you and your batchmates — real usage = real credibility

1.3 Academic Framing (for your report/presentation)

Wrong: "I built an app to track job applications."

Right: "I designed and deployed a RESTful web service with JWT-authenticated endpoints, a normalized relational schema with indexed foreign keys, a layered service architecture, and an analytics engine that computes application funnel metrics — packaged as a personal productivity platform for campus placement management."

Same project. Completely different positioning. Learn to speak the language of backend engineering when describing your work to interviewers and professors.

2 Feature Scope — Build This vs. Skip This

Scope creep is the #1 reason student projects fail. Every feature idea that arises after Week 1 goes into a "Future Work" document — not the codebase. Your grade (and your sanity) depends on a working, polished MVP, not a feature-bloated skeleton.

☑ BUILD THIS (MVP Scope)	⊘ SKIP THIS (Out of Scope)
• JWT authentication (register, login, logout)	• OAuth / Google / LinkedIn sign-in
• Add / Edit / Delete job applications	• Email notifications or reminders (cron jobs)
• Status workflow: Applied → OA → Interview → Offer → Rejected	• Real-time updates (WebSockets)
• Notes field per application	• File uploads (resume PDF storage)
• Filter by status, company, date range	• Multi-user / team collaboration
• Sort by date, company, status	• Role-based access (admin, recruiter)
• Dashboard: total apps, offer rate, rejection rate	• AI-powered resume matching
• Monthly application count (bar chart data)	• Calendar integration (Google / Outlook)
• Export all applications as CSV	• Mobile app (React Native)
• Pagination on the applications list	• Docker / Kubernetes deployment
• Input validation + meaningful error messages	• Microservices architecture
• Basic rate limiting on auth routes	• GraphQL API

3 Technical Stack — Every Tool Explained

3.1 Backend

Technology	What It Does & Why You're Using It
Node.js	JavaScript runtime on the server. Single language across frontend + backend. Huge ecosystem. npm has a package for everything you need.
Express.js	Minimal web framework for Node. Gives you routing, middleware, and request/response handling. Industry standard for REST APIs — every interviewer knows it.
MySQL	Relational database. Perfect for structured data with relationships (users → applications → notes). SQL queries are a core interview topic. Free, fast, and well-documented.
mysql2 (npm)	The best Node.js MySQL driver. Supports async/await natively. Faster than the original mysql package. Use prepared statements to prevent SQL injection.
JSON Web Tokens (jsonwebtoken)	Stateless authentication — no server-side sessions. Token is signed with a secret, verified on every protected request. The standard for REST API auth in 2024.
bcryptjs	Password hashing library. NEVER store plain text passwords. bcrypt adds a salt and applies a slow hash — makes brute-force attacks computationally infeasible.
express-validator	Middleware for validating and sanitizing incoming request data. Catches bad input before it touches your database layer.
express-rate-limit	Prevents brute-force login attacks by limiting requests per IP per time window. Essential for any auth endpoint.
cors	Express middleware to allow or restrict which origins can call your API. Required if your frontend is on a different domain/port.
dotenv	Loads environment variables from a .env file. Keeps secrets (DB password, JWT secret) out of your source code. Never commit .env to Git.
json2csv	Converts a JavaScript array of objects to a CSV string. Powers your Export CSV feature in one line of code.

3.2 Frontend (Minimal — this project is backend-heavy)

Technology	Role
Vanilla HTML/CSS + Fetch API	Keep it simple. No React, no Angular. A few static HTML pages with JavaScript fetch() calls to your API. The focus is on your backend — interviewers will respect this clarity.
Chart.js (CDN)	Renders the dashboard bar/pie charts client-side. One script tag, no build tooling needed.

3.3 Infrastructure & Dev Tools

Tool	Purpose
Railway.app or Render.com	Free cloud hosting for your Node.js app. Connect GitHub → auto-deploys on every push. No server setup needed.
PlanetScale or Aiven MySQL	Free managed MySQL in the cloud. No local MySQL setup required for deployment. Connection string goes in your .env file.
Postman	Test every API endpoint before building the frontend. Treat this as your primary development tool — not the browser.
Git + GitHub	Version control. Clean commit history with meaningful messages demonstrates professionalism. Keep a public repo for your resume.
Nodemon	Auto-restarts your Node server on file changes during development. Install as devDependency only.

4 Database Schema Design

4.1 Schema Overview

Three tables. Clean, normalized, with proper foreign key relationships. This is what a senior engineer would design for this feature set.

4.2 Table: users

Column	Type & Constraints	Purpose
id	INT UNSIGNED AUTO_INCREMENT PRIMARY KEY	Unique user identifier
name	VARCHAR(100) NOT NULL	Display name
email	VARCHAR(255) NOT NULL UNIQUE	Login identifier — UNIQUE enforces no duplicates
password_hash	VARCHAR(255) NOT NULL	bcrypt hash of password — NEVER plain text
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Account creation time

4.3 Table: applications

Column	Type & Constraints	Purpose
id	INT UNSIGNED AUTO_INCREMENT PRIMARY KEY	Unique application ID
user_id	INT UNSIGNED NOT NULL, FK → users(id)	Links application to its owner — with ON DELETE CASCADE
company_name	VARCHAR(150) NOT NULL	Company being applied to
role_title	VARCHAR(150) NOT NULL	e.g. "Software Engineer Intern"
status	ENUM('Applied','OA','Interview','Offer','Rejected') NOT NULL DEFAULT 'Applied'	Current stage in the hiring funnel
applied_date	DATE NOT NULL	When the application was submitted
source	VARCHAR(100)	Where found: LinkedIn, Naukri, Campus, Referral, etc.
salary_lpa	DECIMAL(6,2)	Offered/expected CTC in LPA — nullable until offer stage
notes	TEXT	Free-form notes: interview feedback, deadlines, contacts

Column	Type & Constraints	Purpose
updated_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	Auto-updates on every row change
created_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Row creation time

KEY INDEXES TO CREATE: INDEX idx_user_status ON applications(user_id, status) — used by filter queries. INDEX idx_user_date ON applications(user_id, applied_date) — used by sorting and dashboard monthly aggregation. These two indexes alone will make your queries fast enough to discuss at interviews.

4.4 Table: status_history (Audit Log — shows engineering maturity)

Column	Type & Constraints	Purpose
id	INT UNSIGNED AUTO_INCREMENT PRIMARY KEY	Row ID
application_id	INT UNSIGNED NOT NULL, FK → applications(id)	Which application changed
from_status	VARCHAR(50)	Previous status value (NULL for first entry)
to_status	VARCHAR(50) NOT NULL	New status value
changed_at	TIMESTAMP DEFAULT CURRENT_TIMESTAMP	When the change happened

Why this table matters: It proves you think about data history, not just current state. You can tell the interviewer: 'Every time an application status changes, we log the transition. This gives us a complete audit trail and enables features like time-in-stage analytics.' That is senior-level thinking.

4.5 Entity Relationship

Relationship	Type	Constraint
users → applications	One-to-Many (1 user has many applications)	ON DELETE CASCADE — deleting a user clears all their data
applications → status_history	One-to-Many (1 application has many history rows)	ON DELETE CASCADE — clean deletion

5 API Design — Complete Endpoint Reference

Every endpoint follows the same pattern: authenticate → validate input → call service layer → return JSON. No business logic lives in route handlers.

5.1 Auth Endpoints (no JWT required)

Method + Path	Request Body	Response	Notes
POST /api/auth/register	{ name, email, password }	{ message, token, user }	Hash password with bcrypt rounds=12. Return JWT immediately.
POST /api/auth/login	{ email, password }	{ message, token, user }	Compare hash with bcrypt.compare(). Return 401 on failure.
GET /api/auth/me	— (JWT in header)	{ user }	Verify token, return current user profile.

5.2 Application Endpoints (JWT required on all)

Method + Path	Query Params / Body	Response
GET /api/applications	?status=&company=&sort=date_desc&page=1&limit=20	{ data: [...], total, page, totalPages }
POST /api/applications	{ company_name, role_title, status, applied_date, source, notes }	{ message, data: newApplication }
GET /api/applications/:id	—	{ data: application }
PUT /api/applications/:id	Any updatable fields	{ message, data: updatedApplication }
DELETE /api/applications/:id	—	{ message }
GET /api/applications/:id/history	—	{ data: [statusHistoryRows] }
PATCH /api/applications/:id/status	{ status: 'Interview' }	{ message, data } — also writes to status_history

5.3 Dashboard & Export Endpoints (JWT required)

Method + Path	Response
GET /api/dashboard/summary	{ total, byStatus: {Applied:N, OA:N, ...}, offerRate, avgResponseDays }
GET /api/dashboard/monthly	{ data: [{month:'Jan 2025', count:12}, ...] } — last 6 months
GET /api/applications/export/csv	Content-Type: text/csv — file download with all user's applications

Standard Response Envelope: All success responses return { success: true, data: {...} }. All error responses return { success: false, error: 'message', code: 'ERROR_CODE' }. This consistency is what separates a professional API from a student project.

6 Backend Architecture — Folder Structure & Layers

6.1 Layered Architecture

The single most important architectural decision you will make. Route handlers do NOT contain business logic. Service functions do NOT contain SQL. This separation is what every backend engineer means by 'clean architecture'.

Layer	File Location	Responsibility
Route Handler	routes/applications.js	Parse HTTP request, call validator, call service, return HTTP response. Nothing else.
Validator	validators/applicationValidator.js	express-validator chains — check required fields, enums, types. Run before any service call.
Service	services/applicationService.js	Business logic — ownership check, status transition rules, calling repository functions.
Repository	repositories/applicationRepository.js	All SQL queries live here. Returns raw JS objects. The ONLY place that touches MySQL.
Middleware	middleware/auth.js	JWT verification — attaches req.user to every protected request.
DB Config	config/db.js	MySQL connection pool setup. One pool shared across the entire app.

6.2 Complete Folder Structure

```
tracker-backend/
├── config/
│   └── db.js                ← MySQL pool (mysql2/promise)
├── middleware/
│   ├── auth.js              ← JWT verify → req.user
│   └── errorHandler.js      ← Global error handler
├── routes/
│   ├── auth.js
│   ├── applications.js
│   └── dashboard.js
├── validators/
│   ├── authValidator.js
│   └── applicationValidator.js
├── services/
│   ├── authService.js
│   └── applicationService.js
├── repositories/
│   ├── userRepository.js
│   └── applicationRepository.js
```

```
├── utils/
│   └── csvExport.js
├── public/                    ← Static frontend (HTML/CSS/JS)
│   ├── index.html
│   ├── dashboard.html
│   └── js/
├── .env                       ← NEVER commit this
├── .env.example               ← DO commit this
├── .gitignore
├── app.js                     ← Express app setup
└── server.js                  ← Entry point (listen on PORT)
```

7 4-Week Execution Plan

Each week has one owner and one deliverable you can demo. Do not start Week 2 tasks until Week 1 is fully working and tested in Postman.

WEEK 1

Foundation — Setup, Auth & Schema

→ Install Node.js, MySQL (local), Git, Postman, VSCode
→ npx express-generator --no-view or scaffold manually
→ Set up MySQL database + run CREATE TABLE scripts for all 3 tables
→ Build config/db.js connection pool with mysql2/promise
→ Build POST /api/auth/register — hash password, insert user, return JWT
→ Build POST /api/auth/login — compare hash, return JWT
→ Build auth.js middleware — verify JWT, attach req.user
→ Build GET /api/auth/me — protected, returns user profile
→ Write a 20-line test script in Postman — all 3 auth routes pass
→ Push to GitHub with .env.example and meaningful README

Deliverables

✓ Running Express server with MySQL connected
✓ 3 auth endpoints working in Postman
✓ JWT tokens generated and verified
✓ GitHub repo with initial commit

WEEK 2

Core CRUD — Applications & Status Workflow

→ Build GET /api/applications — with user_id filter, pagination (LIMIT/OFFSET)
→ Build POST /api/applications — validate input, insert row, return new record
→ Build GET /api/applications/:id — ownership check (user_id must match req.user.id)
→ Build PUT /api/applications/:id — partial update, ownership check
→ Build DELETE /api/applications/:id — ownership check before delete
→ Build PATCH /api/applications/:id/status — update status + INSERT into status_history
→ Build GET /api/applications/:id/history — return status_history rows
→ Add query param filtering: ?status=Applied&company=Google&sort=date_desc
→ Add express-validator on POST and PUT routes
→ Test all 8 endpoints thoroughly in Postman — edge cases included

Deliverables

✓ Full CRUD for applications working
✓ Status workflow with audit log
✓ Filtering + sorting functional
✓ All routes protected by JWT middleware

WEEK 3

Dashboard, Export & Frontend

→ Build GET /api/dashboard/summary — COUNT and GROUP BY status in one SQL query

→ Build GET /api/dashboard/monthly — GROUP BY YEAR(applied_date), MONTH(applied_date)

→ Build GET /api/applications/export/csv — use json2csv, set Content-Disposition header

→ Add express-rate-limit to /api/auth/* routes (max 10 req/15min)

→ Add a global errorHandler.js middleware — catch all unhandled errors

→ Build the HTML frontend: login page, applications list page, add/edit form

→ Build the dashboard page with Chart.js bar chart using API data

→ Connect frontend fetch() calls to your API endpoints with JWT in Authorization header

→ Deploy backend to Railway.app or Render.com — test live URL

→ Deploy frontend (same server, serving /public as static files)

Deliverables

✓ Dashboard analytics API live

✓ CSV export downloading correctly

✓ Frontend connected to your real API

✓ App live on public URL

WEEK 4

Polish, Security & Submission

→ Audit all routes: ensure every endpoint has input validation

→ Add SQL injection test — verify prepared statements protect all queries

→ Add helmet.js (sets security HTTP headers in one line)

→ Write a test checklist: 15 scenarios (invalid token, wrong user, missing fields, etc.)

→ Benchmark two key queries with EXPLAIN — verify indexes are being used

→ Write the project report: Architecture section + API docs + DB schema diagram

→ Create system architecture diagram on draw.io: Browser → Express → Service → Repo → MySQL

→ Record a 3-minute demo video: register → add apps → change status → view dashboard → export CSV

→ Clean up GitHub: meaningful commits, no console.logs, .env never committed

→ Prepare 5 answers for likely interview questions about this project

Deliverables

✓ Security hardened — helmet, rate-limit, validation

✓ EXPLAIN confirms indexes used

✓ Architecture diagram complete

✓ Demo video recorded

✓ Interview-ready talking points

8 Risk Mitigation

Risk	Severity	Mitigation
MySQL connection pool exhausted under concurrent requests	Medium	Set connectionLimit: 10 in mysql2 pool config. For a personal app, this is more than enough. Use SHOW PROCESSLIST if debugging.
JWT secret exposed in code	HIGH	Always load from process.env.JWT_SECRET via dotenv. Add .env to .gitignore. Use .env.example with placeholder values. Check with git log before any public push.
SQL injection attack	HIGH	Use prepared statements (parameterized queries) for every user-supplied value. Never concatenate user input into SQL strings. mysql2 makes this easy: db.query('SELECT * FROM applications WHERE id = ?', [id])
User A can read/edit User B's applications	HIGH	Every query includes WHERE user_id = req.user.id. The service layer does this check before any DB call — tested explicitly in your test checklist.
Render.com spins down on inactivity (free tier)	Medium	Use UptimeRobot (free) to ping your server every 5 minutes. Or mention in your presentation that production would use a paid tier — professors understand this.
Forgot to handle async errors in Express	Medium	Wrap every async route handler in a try/catch or use an asyncHandler utility wrapper. Add a global error middleware as the last app.use() in app.js.
Database data lost if PlanetScale free tier closes	Low	Export a SQL dump weekly during development. For the demo, have a seed script that repopulates test data in 30 seconds.

9 Metrics to Measure & Present

Professors and interviewers need numbers. Here is exactly what to measure and how — all achievable without cloud infrastructure.

9.1 Performance Benchmarks

Metric	Target	How to Measure
GET /api/applications response time	< 100ms with 500 rows	Add console.time/timeEnd around DB call. Seed 500 rows, time in Postman.
POST /api/auth/login response time	< 300ms	bcrypt is intentionally slow (that is the security). 300ms is acceptable and expected.
MySQL query time for filtered list	< 10ms	Run EXPLAIN SELECT on your filter query. Confirm 'Using index' in the output.
Dashboard summary query time	< 20ms	Single aggregation query with GROUP BY status — log the query execution time.
CSV export for 200 records	< 500ms	Time the full request in Postman. 200 rows through json2csv is near-instant.
App load time (first page)	< 2s on 4G	Use browser DevTools Network tab. Most of your load time will be the Render.com cold start.

9.2 Security Checklist — Show This in Your Report

Security Control	Implementation	Status
Password hashing	bcrypt with 12 rounds	Required
JWT expiry	expiresIn: '7d' — tokens expire automatically	Required
Input validation	express-validator on all POST/PUT routes	Required
Prepared statements	All SQL uses parameterized queries via mysql2	Required
Ownership check	Every query includes WHERE user_id = req.user.id	Required
Rate limiting	express-rate-limit on /api/auth/* (10 req/15min)	Required
Security headers	helmet() added to app.js in one line	Required
Secrets management	.env for all secrets, never committed to Git	Required

10 Future Enhancements — Shows Depth of Thinking

A dedicated Future Work section shows professors and interviewers that you understand the full problem space, not just the code you wrote. Include these in your report and presentation.

Email Reminders (Notifications Layer)

Integrate Nodemailer + cron (node-cron package) to send weekly digest emails: 'You have 3 applications in Interview stage that haven't moved in 7 days.' Technically: cron job queries status_history for stale applications, calls email service. Requires SMTP credentials in .env.

Advanced Analytics

Expand the dashboard to show: average days between status transitions (time-in-stage), source effectiveness (which job board produces the most interviews), offer rate by company tier. All computable with SQL window functions on the status_history table.

Google OAuth / LinkedIn Login

Replace password login with OAuth 2.0 via Passport.js. Users click 'Continue with Google', no password needed. The users table gains an oauth_provider and oauth_id column. Eliminates the need to store password hashes entirely.

Resume Attachment

Allow users to upload a resume PDF per application. Backend stores files in AWS S3 or Cloudinary free tier. The applications table gains a resume_url column. Requires multipart/form-data parsing with multer middleware.

Mobile-Responsive Frontend → React App

Replace the vanilla HTML frontend with a React SPA using React Query for data fetching. The backend API requires zero changes — this is the power of building a clean REST API first. Deploy frontend on Vercel (free tier) separately from the backend.

Deadline Alerts

Add an application_deadline DATE column to the applications table. A daily cron job finds applications where deadline is within 48 hours and status is still 'Applied' — then pushes browser notifications or sends email alerts. Simple to implement once the email layer exists.

Closing Reminder: You have a complete, interview-ready blueprint. The architecture is intentionally simple enough to finish in 4 weeks but engineered well enough to discuss in depth at any backend interview. Build it, deploy it, and put the live URL on your resume. Good luck.

— End of Document —