CSCI 201 Final Project Documentation

Job Tracker

Aditi Jain
Aiden Howard
Annie Gao
Alex Welch
Dylan Sun
Leyaa George
Vanessa Gonzalez
Yuvika Poddar

Table of Contents

Project Proposal

JobTracker is a full-stack job application management system built with a Java Spring Boot backend and HTML/CSS/JavaScript frontend, using PostgreSQL (Supabase) for data persistence. The application enables users to track job applications across multiple companies, with additional features for Gmail and Google calendar users.

High-Level Requirements

Automation and Tracking

- The system should remind users of important dates such as application deadlines, interviews, and follow-ups through a calendar view and timeline features.
- It should connect with a user's calendar to automatically add upcoming events and reminders.
- The program should recognize emails related to job applications and update the user's progress accordingly through Gmail integration.
- The system should allow users to save and view key details for each application, including job links, interview dates/times, meeting links, locations, salary ranges, and notes.
- Users should be able to track application progress through multiple status stages: Draft, Applied, Interview, Offer, and Rejected.
- The system should maintain a timeline of events for each application showing deadlines, interviews, and other important dates.

Job Research and Recommendations

- The program should fetch job listings from external job APIs and store them in the database.
- The system should allow users to search and filter through available job listings.
- Users should be able to quickly create new applications from recommended jobs by pre-filling application forms with job details.
- The system should refresh job listings automatically on a daily schedule and on initial startup if the database is empty.

User Management

- The system should support user registration and authentication using JWT tokens.
- Users should be able to securely log in and manage their own job applications independently.
- The system should provide session management with token expiration and refresh capabilities.

Integration Capabilities

- The system should support Google OAuth integration for Gmail and Calendar access.
- Users should be able to enable/disable Gmail scanning and calendar sync features independently.
- The system should run background scans of Gmail when enabled, processing only new emails to update application statuses automatically.
- The system should sync application deadlines and interview events to the user's Google Calendar when enabled.

Technical Specifications

1. Core Functional Requirements

1.1 User Authentication and Registration

Requirements:

- Enable user registration and login functionality.
- Provide different interfaces for guest and authenticated users.

Implementation:

- Backend: Java Spring Boot 3.2.3 REST API with Spring Security.
- Database: PostgreSQL via Supabase (users table with id, username, email, password_hash, role, created_at, updated_at).
- Security: Passwords hashed using BCrypt; JWT-based authentication with refresh tokens.
- Frontend: HTML, CSS, JavaScript login and registration pages with modern gradient design.
- Networking: HTTPS communication via REST API endpoints.

API Endpoints:

Method	Endpoint	Description	Request	Response
POST	/api/auth/register	Register new user	{email, password, username}	201 Created with {jwt, refreshToken, expiresAt, refreshExpiresAt, username, email, role} / 409 Conflict
POST	/api/auth/login	Authenticate user	{email, password}	200 OK with {jwt, refreshToken, expiresAt, refreshExpiresAt, username, email, role}

POST	/api/auth/logout	Logout user	Authorization Header	200 OK with {message}

Acceptance Criteria:

- Unauthenticated users cannot access protected endpoints (/api/apps/**, /api/user/**).
- Passwords are securely stored as BCrypt hashes.
- JWT tokens expire after 15 minutes (900000ms); refresh tokens are valid for 7 days (604800000ms).
- Client-side token validation prevents access to protected pages.

Rubric Alignment:

- Fulfills Java backend, authentication, and networking requirements.

1.2 Job Application Tracking and Automation

Requirements:

- Allow users to track job applications, deadlines, interviews, and follow-ups.
- Provide manual reminder creation and calendar event management.
- Support automatic Gmail scanning for application status updates.

Implementation:

- Backend: Spring Boot REST API with @EnableScheduling for background tasks.
- Database: Tables for applications, reminders, and email_sync_state.
- Scheduler: Spring's @Scheduled annotation for background jobs (GmailBackgroundScanner, JobScheduler).
- Gmail Integration: Google Gmail API (OAuth2) for email scanning.
- Calendar Integration: Google Calendar API (OAuth2) - planned but not yet implemented.
- Frontend: HTML, CSS, JavaScript dashboard, application details, and calendar views.

API Endpoints:

Method	Endpoint	Description
GET	/api/apps	Retrieve user's applications
POST	/api/apps	Create new application

GET	/api/apps/{id}	Get specific application
PUT	/api/apps/{id}	Update application
PUT	/api/apps/{id}/status	Update application status only
DELETE	/api/apps/{id}	Delete application
GET	/api/reminders	Get all reminders
POST	/api/reminders	Create reminder
GET	/api/reminders/{id}	Get specific reminder
PUT	/api/reminders/{id}	Update reminder
DELETE	/api/reminders/{id}	Delete reminder
GET	/api/dashboard-summary	Get dashboard metrics

Threading Model:

- GmailBackgroundScanner: @Scheduled task executes every 60 seconds (60000ms fixed rate).
- JobScheduler: @Scheduled cron job runs daily at 8:00 AM and on startup if database is empty.
- Thread Safety: Background jobs use transactional database operations; sync state prevents duplicate email processing.

Acceptance Criteria:

- Gmail scanning runs every 60 seconds when Google integration is enabled and connected.
- Email sync state tracks last processed internalDate to prevent duplicate processing.
- Application statuses automatically update based on email content keywords.
- Manual reminders can be created for deadlines, interviews, and follow-ups.
- Auto-generated deadline events appear in timeline without duplicating manually created reminders.

Rubric Alignment:

- Demonstrates required networking and multithreading in Java.

1.3 Email Parsing and Progress Updates

Requirements:

- Automatically detect job-related emails and update application status.

Implementation:

- Integration: Gmail API (OAuth2) via google-api-services-gmail v1-rev20220404-1.32.1.
- Authentication: Google OAuth flow with offline access and token storage in local file system.
- Parsing: EmailParserService with regex-based keyword extraction using patterns for interview, offer, rejection, and application confirmation.
- Threads: @Scheduled background task (GmailBackgroundScanner) with 60-second fixed rate.
- Database: Auto-update status field in applications table; track sync state in email_sync_state table.
- Matching: Applications matched by company name in email sender or subject.

Acceptance Criteria:

- Scanning occurs asynchronously via Spring's scheduling framework.
- Duplicate processing prevented via internalDate tracking in email_sync_state table.
- Status updates correctly mapped to application records based on company name matching.
- Notes field auto-updated with timestamp and new status on each email-triggered change.
- Gmail integration can be enabled/disabled via settings page.

1.4 Job Research and Recommendations

- Requirements:
- • Provide job suggestions based on an external job API.
- • Expose endpoints that return stored jobs and ranked recommendations based on user skills/keywords for the frontend to consume.

Implementation:

- APIs: Adzuna Job Search API using app_id and app_key from configuration (environment variables / application.properties).

- Logic: JobRecommendationService performs simple content-based scoring by counting keyword matches from the user's skills in the job title and description, then sorting by score.
- Database: jobs table mapped to the Job entity with fields: title, company, salary, description, location, and external_url.
- Backend:
- AdzunaClient calls the Adzuna API and converts responses into Job entities.
- JobRepository persists jobs in the jobs table.
- JobRecommendationController exposes REST endpoints to fetch jobs from Adzuna and to return recommendations.

API Endpoints:

Method	Endpoint	Description
POST	/api/jobs/fetch	Fetch jobs from Adzuna (query + location) and store them in the database
POST	/api/jobs/recommendations	Return recommended jobs scored by skill keyword matching

Acceptance Criteria:

- API calls retrieve and store jobs successfully from Adzuna via /api/jobs/fetch.
- Recommendation scoring is based on keyword matching in job title and description.
- /api/jobs/recommendations returns ranked JobDto results that align with user-provided skills.
- Stored jobs persist in the database

1.5 Graphical User Interface (GUI)

Requirements:

- Modern, responsive, and user-friendly web interface.
- Exclude JavaFX, Swing, or JSP.

Implementation:

- Framework: Vanilla HTML, CSS, JavaScript (no frontend framework).
- Styling: Custom CSS with theme support (light/dark mode).
- Core Pages:
    - Login / Register (gradient orb animations)
    - Dashboard (metrics and upcoming events)
    - Applications List (search, filters, status chips)
    - Application Details (timeline, status dropdown, notes)

    - Calendar View (month/week/day views with event modals)
    - Job Recommendations (search and quick apply)
    - Settings/Integrations (Google OAuth, theme toggle)
- Components: Reusable navbar loaded dynamically, modal system for event creation/editing.

Acceptance Criteria:

- Frontend communicates with backend via REST APIs using JWT authentication.
- Interface follows consistent styling with status chips, modern buttons, and card layouts.
- Responsive design works across different screen sizes.
- Theme preference persists in localStorage.
- Authentication state managed client-side with token expiration checking.

---

2. Database Schema

Table	Key Fields	Description
users	id (PK), created_at, email, password_hash, role, updated_at, username	Stores user authentication info
applications	id (PK), user_id (FK), applied_at, company, created_at, deadline_at, experience, interview_at, job_link, job_type, location, notes, salary, status, title	Core user applications
reminders	id (PK), application_id (FK), color, created_at, end_date, end_time, kind, location, meeting_link, notes, sent_at, start_time, title, trigger_at	Notification tracking
jobs	id (PK), company, description, external_url, location, salary, title	Job research listings
email_sync_state	id (PK), last_processed_internal_date	Tracks last processed email ID

---

3. Networking and Security

Networking Overview:

- Client (HTML, CSS, JavaScript) communicates with server via REST API over HTTP (port 8080 for development).
- Server connects to Google APIs (Gmail) using OAuth2 with offline access.
- Server connects to Adzuna API using app_id and app_key query parameters.
- Token storage: Google OAuth credentials stored in local file system (`tokens` directory).
- CORS: Configured to allow all origins (*) for development via CorsConfig and SecurityConfig.

Security Measures:

- Passwords are hashed with BCrypt via Spring Security's BCryptPasswordEncoder.
- JWT-based authentication using io.jsonwebtoken (jjwt) library version 0.12.3.
- Access tokens expire after 15 minutes (900000ms); refresh tokens valid for 7 days (604800000ms).
- JWT secret configurable via environment variable or default value in application.properties.
- HTTPS communication in production (configured via server.port and context-path).
- Spring Security filter chain with JwtAuthenticationFilter for token validation.
- Protected endpoints require valid JWT in Authorization header (Bearer token).
- Public endpoints: /api/auth/register, /api/auth/login, /api/oauth/google/**, static resources.
- CSRF disabled for stateless API (SessionCreationPolicy.STATELESS).
- Input validation through Spring's @Valid annotations and JPA constraints.

API Security Configuration:

- Google OAuth: credentials.json file required in src/main/resources.
- Adzuna API: app_id and app_key configured in application.properties or environment variables.
- Database credentials: PostgreSQL connection via Supabase with SSL required mode.
- Sensitive files excluded via .gitignore: credentials.json, tokens directory.

---

4. Testing and Deployment

Testing Plan:

- Unit Testing: JUnit available via spring-boot-starter-test dependency.
- Integration Testing: Spring MockMvc available for endpoint validation.
- Manual Testing: Frontend UI testing through browser.
- Data Management: Database schema auto-updated via Hibernate (spring.jpa.hibernate.ddl-auto=update).

Current Deployment Configuration:

- Database: PostgreSQL hosted on Supabase (aws-0-us-west-2.pooler.supabase.com:6543).
- Backend: Spring Boot application (port 8080).
- Static Resources: Served from src/main/resources/static directory.
- Build Tool: Maven with spring-boot-maven-plugin.
- Java Version: 17 (configured in pom.xml and Eclipse settings).

Database Configuration:

- Connection pooling: HikariCP with PgBouncer compatibility settings.
- SSL Mode: Required for Supabase connection.
- Prepared statement caching disabled for transaction pooler compatibility.
- Schema management: Automatic via Hibernate with spring.jpa.hibernate.ddl-auto=update.
- SQL logging: Enabled via spring.jpa.show-sql=true for debugging.

Development Setup:

- IDE: Eclipse project configuration included (.project, .classpath, .settings/).
- Dependencies: Managed via Maven (pom.xml).
- Hot reload: Spring Boot DevTools available.
- API Keys Required:
    - Google OAuth: credentials.json file
    - Adzuna: ADZUNA_APP_ID and ADZUNA_APP_KEY environment variables
    - JWT: JWT_SECRET environment variable (optional, has default)

Runtime Requirements:

- Java 17 or higher
- PostgreSQL database (or Supabase connection)
- Google OAuth credentials for Gmail integration
- Adzuna API credentials for job recommendations
- Minimum 512MB RAM recommended

Monitoring and Logging:

- Application logging: SLF4J with Logback (Spring Boot default).
- Scheduled task logging: JobScheduler and GmailBackgroundScanner log execution.
- SQL query logging: Enabled via spring.jpa.show-sql and hibernate.format_sql.
- Error handling: Try-catch blocks in scheduled tasks to prevent silent failures.

---

5. Tools and Technologies Summary

Component	Technology	Free	Purpose
Backend	Java (Spring Boot 3)	Yes	Core API logic
Frontend	HTML, CSS, JavaScript	Yes	User interface
Database	PostgreSQL 16	Yes	Persistent storage
Scheduler	Spring @ Scheduled	Yes	Automated reminders
Email Service	Google Gmail API	Yes	Email scanning
Calendar	Google Calendar API	Yes	Event management

Job Listings	Adzuna API	Yes	Job data
Security	BCrypt, JWT	Yes	Authentication
OAuth	Google OAuth	Yes	Google services authentication
Build Tool	Maven	Yes	Dependency management and build automation
IDE	Eclipse	Yes	Development environment
ORM	Hibernate (via Spring Data JPA)	Yes	Database entity mapping and queries
Connection Pool	HikariCP	Yes	Database connection pooling
JSON Processing	Jackson (via Spring Boot)	Yes	JSON serialization / deserialization
Lombok	Project Lombok	Yes	Boilerplate code reduction
HTTP Client	Rest Template	Yes	External API calls

Detailed Design

Database Design

1. users

Stores authentication and role information for each registered user.

Field	Description
id	BIGINT PRIMARY KEY AUTO_INCREMENT — unique identifier for each user.
email	VARCHAR(255) UNIQUE NOT NULL — user's email address.
username	VARCHAR(255) UNIQUE NOT NULL — user's display name.
password_hash	VARCHAR(255) NOT NULL — securely hashed password using BCrypt.
role	VARCHAR(50) NOT NULL DEFAULT 'USER' — defines access level ('USER' or 'ADMIN').
created_at	TIMESTAMP — record creation timestamp.
updated_at	TIMESTAMP — last update timestamp.

2. applications

Stores each job application created by users, including details and progress.

Field	Description
id	BIGINT PRIMARY KEY AUTO_INCREMENT — unique identifier for each application.
company	VARCHAR(255) — name of the company.
title	VARCHAR(255) — job title applied for.
status	VARCHAR(50) — current application status ('DRAFT', 'APPLIED', 'INTERVIEW', 'REJECTED', 'OFFER').
deadline_at	DATE — application deadline.
interview_at	TIMESTAMP — scheduled interview date and time.
location	VARCHAR(255) — job location.
job_type	VARCHAR(100) — employment type (e.g., 'Full-time', 'Part-time', 'Internship', 'Contract').

salary	VARCHAR(100) — salary range or information.
job_link	VARCHAR(500) — URL to job posting.
experience	VARCHAR(100) — required experience level.
created_at	DATE — record creation date.
applied_at	DATE — date application was submitted.
notes	TEXT — user notes and automatically added status updates from email scanning.

3. reminders

Tracks reminders and notifications related to deadlines, interviews, and follow-ups.

Field	Description
id	BIGINT PRIMARY KEY AUTO_INCREMENT — unique identifier for reminder.
kind	VARCHAR(50) — type of reminder ('DEADLINE', 'INTERVIEW', 'FOLLOWUP').
title	VARCHAR(255) — reminder title/description.
application_id	BIGINT — foreign key referencing applications(id), nullable.
notes	TEXT — additional reminder notes.
color	VARCHAR(50) — color code for calendar display.
created_at	DATE — record creation date.
trigger_at	DATE — date reminder should trigger.
sent_at	DATE — date reminder was sent/processed.
end_date	DATE — end date for interview events.

start_time	VARCHAR(10) — start time for interview events (e.g., "14:00").
end_time	VARCHAR(10) — end time for interview events (e.g., "15:00").
location	VARCHAR(255) — physical location for interview events.
meeting_link	VARCHAR(500) — virtual meeting URL for interview events.

4. jobs

Contains job listings fetched from Adzuna API for recommendations.

Field	Description
id	BIGINT PRIMARY KEY AUTO_INCREMENT — unique identifier for each job.
title	VARCHAR(255) NOT NULL — job title.
company	VARCHAR(255) NOT NULL — company offering the job.
salary	VARCHAR(100) — formatted salary information (e.g., "80k - 120k").
description	TEXT — job description and details.
location	VARCHAR(255) — job location.
external_url	VARCHAR(500) — link to original job posting (redirect URL from Adzuna).

5. email_processing_state

Tracks the last processed email timestamp to prevent duplicate processing of Gmail data.

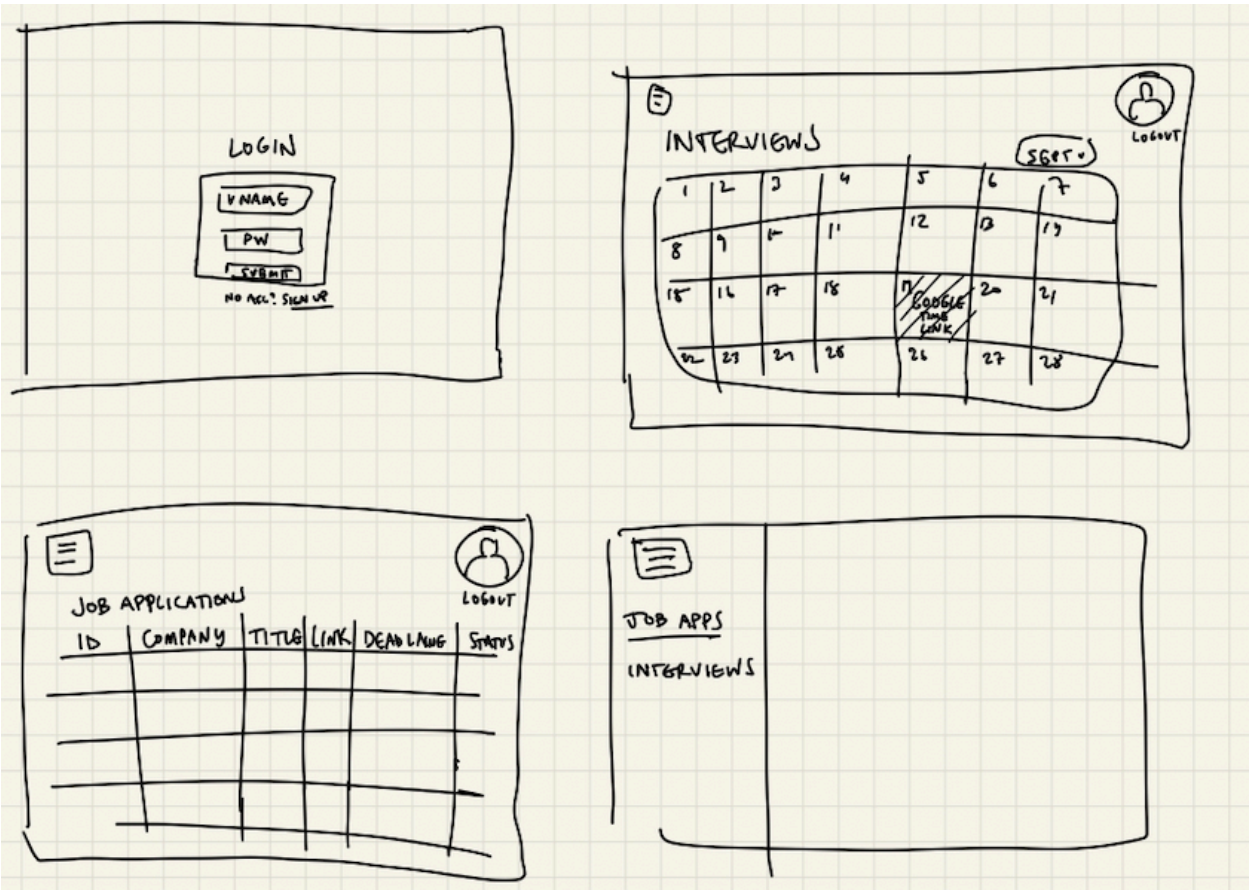
Field	**Description**
id	BIGINT PRIMARY KEY — always set to 1 (single row table).

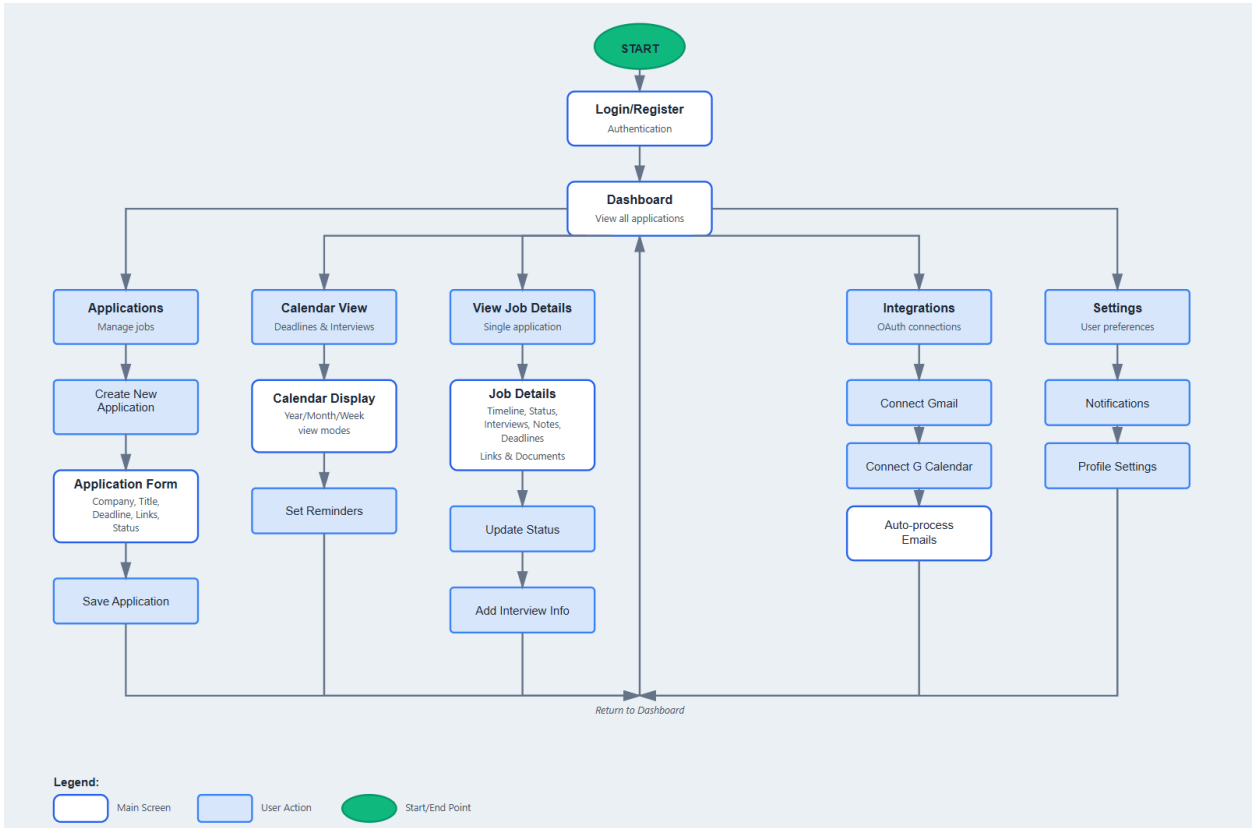
last_processed_internal_date	BIGINT — last processed Gmail internalDate timestamp in milliseconds.

Hardware and software requirements

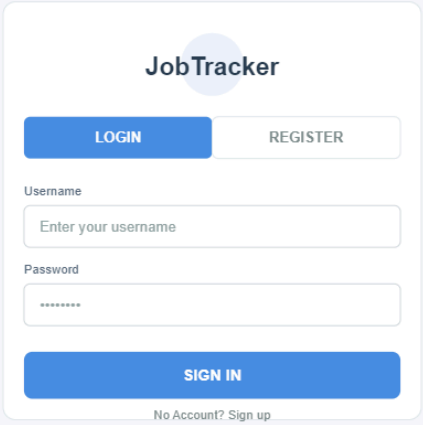
- Hardware:
  - Minimum 512MB RAM recommended for Spring Boot application
  - Storage for PostgreSQL database (provided by Supabase)
  - Internet connection required for external API calls (Google, Adzuna)
- Software
  - Database: PostGre SQL
  - Backend: JDBC, Java
  - Frontend: HTML, CSS, JavaScript

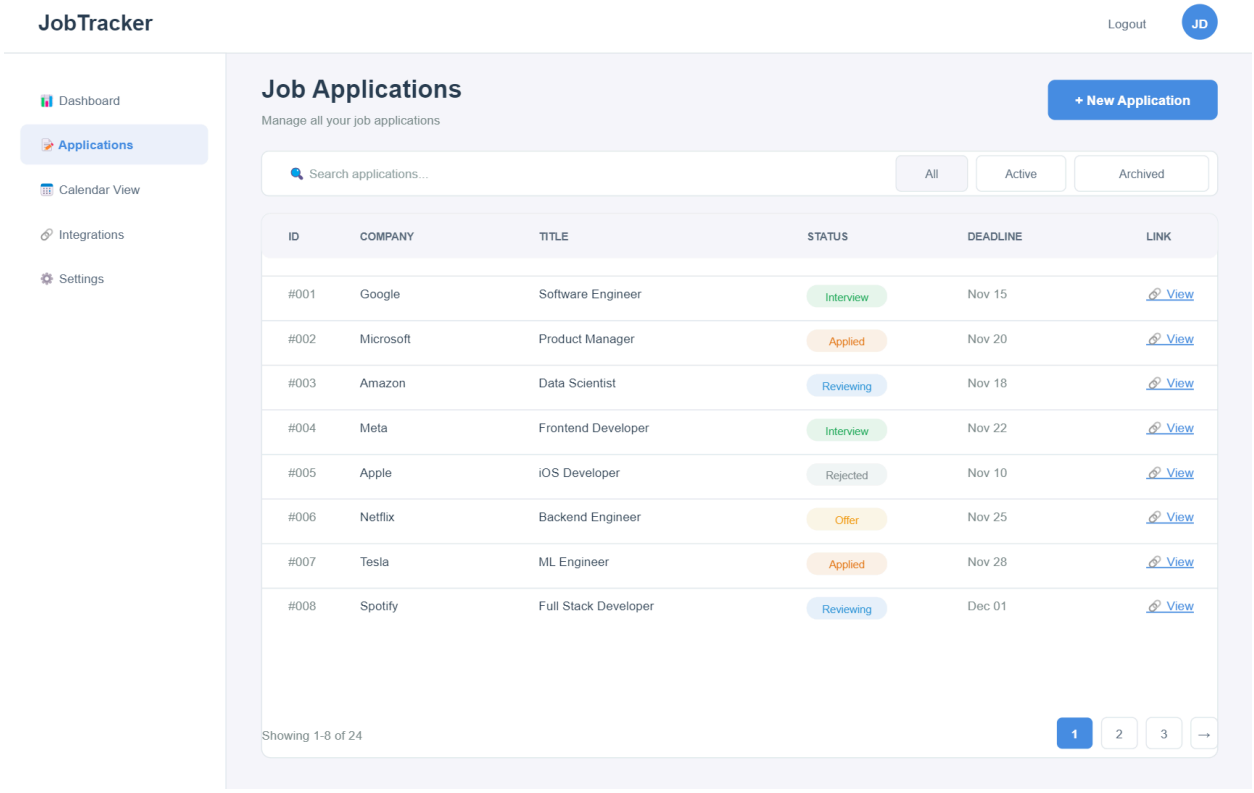
GUI design

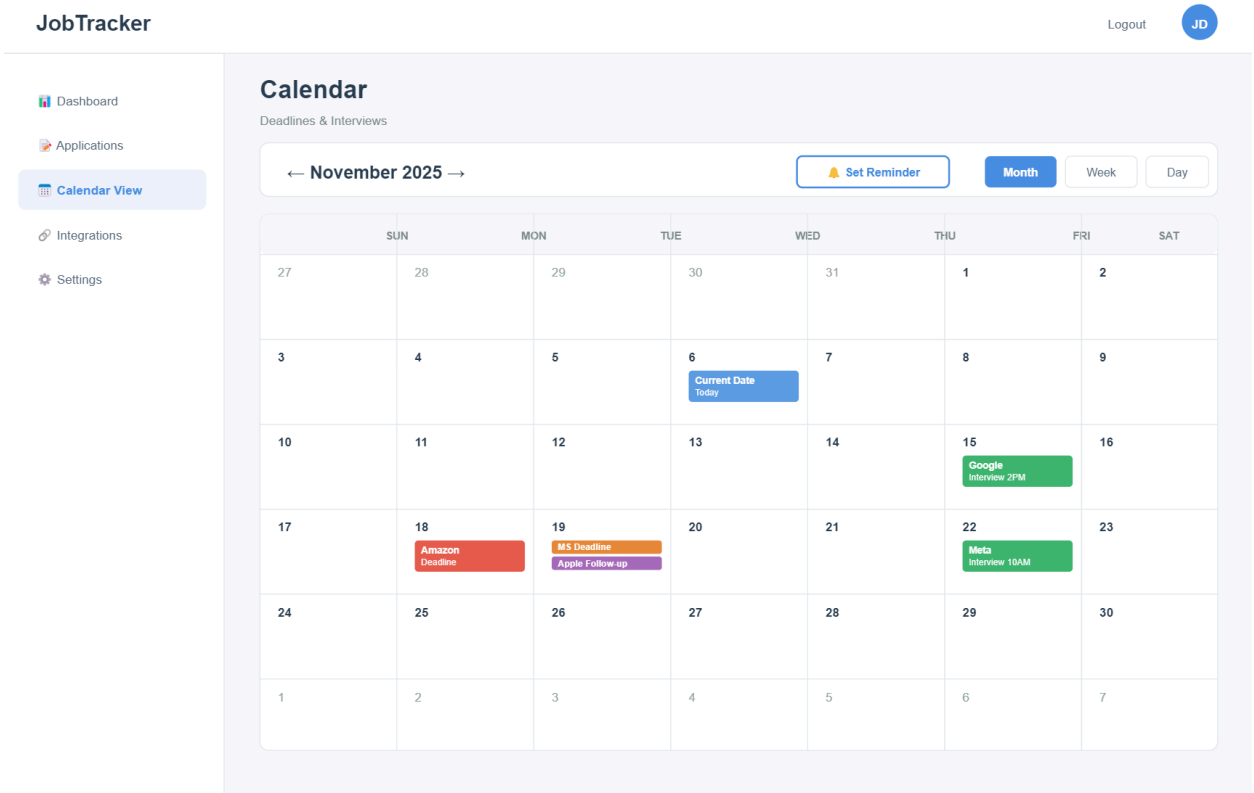

LOGIN
USERNAME
PW
SUBMIT
NO ACC? SIGN UP
INTERVIEWS
SORT
LOGOUT
GOOGLE MEET LINK
JOB APPLICATIONS
LOGOUT
ID
COMPANY
TITLE
LINK
DEADLINE
STATUS
JOB APPS
INTERVIEWS

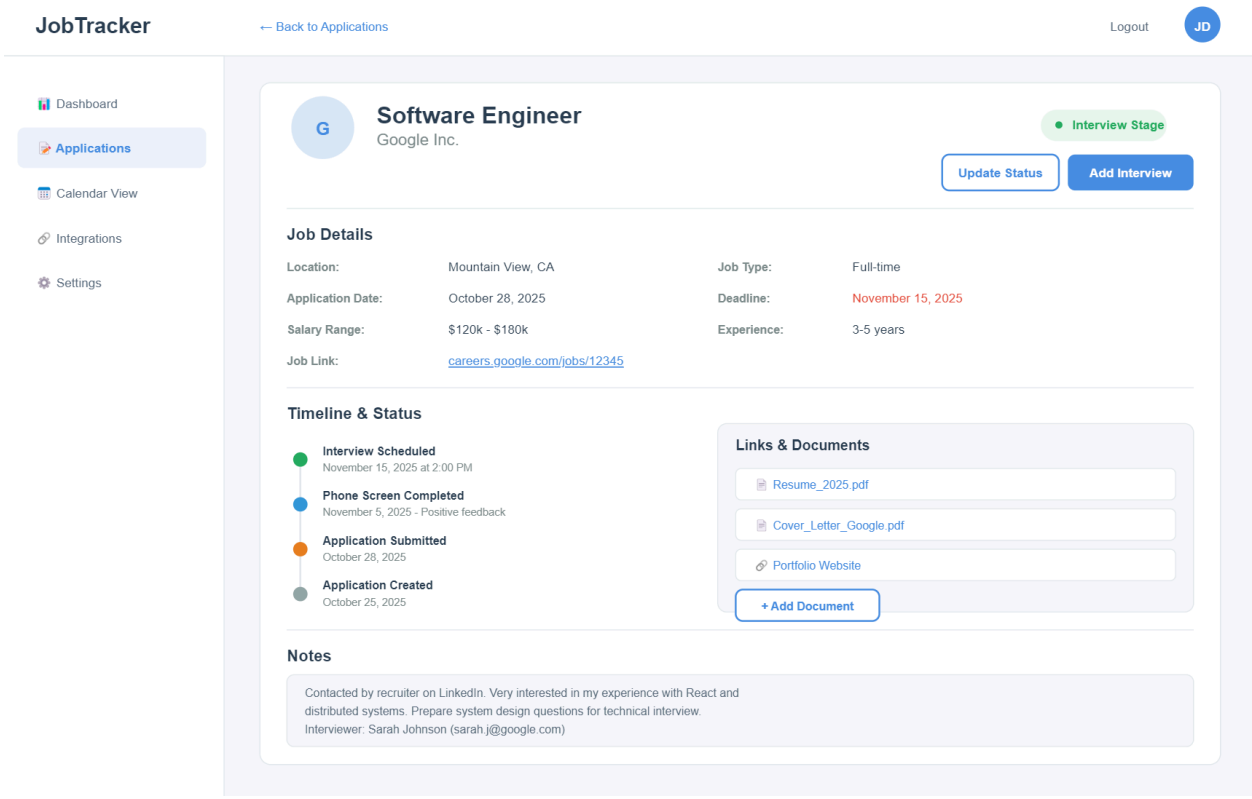


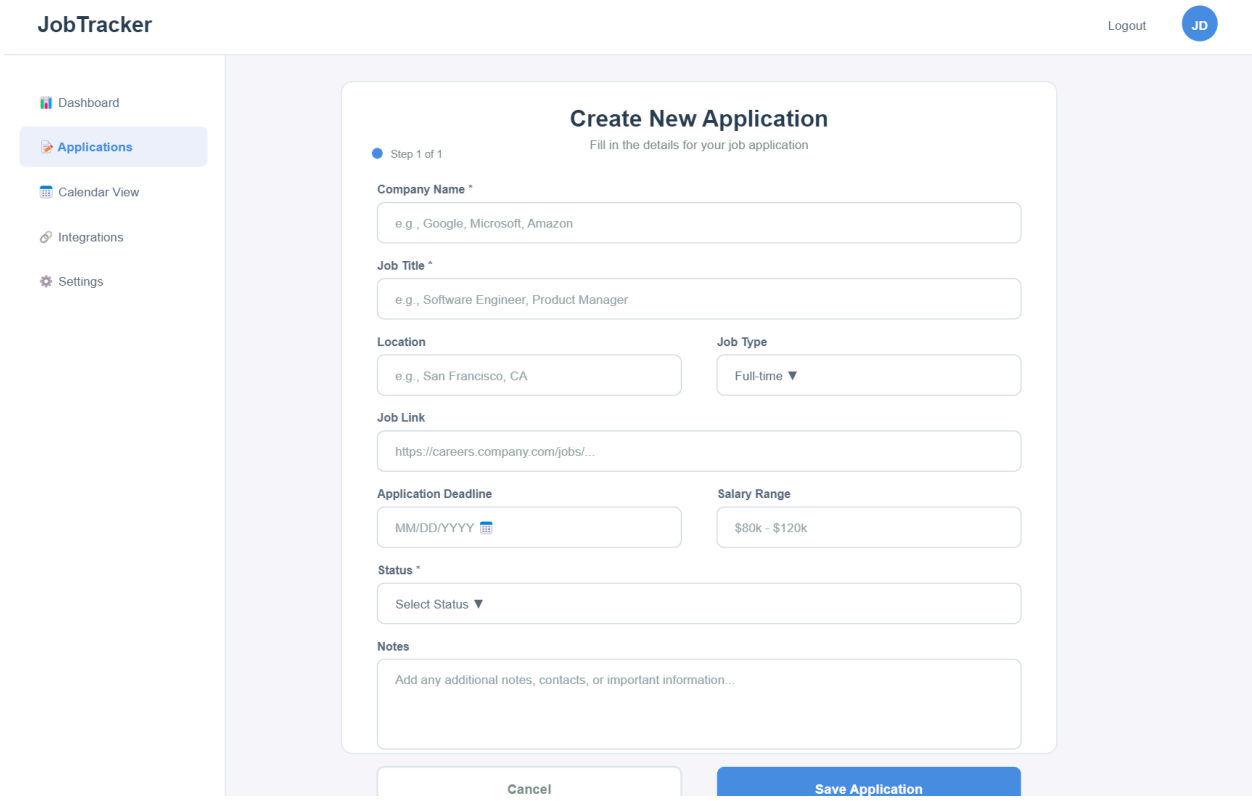
Job Application Tracker - User Flow Diagram




JobTracker
Logout
JD
Dashboard
Applications
Calendar View
Integrations
Settings
Job Applications
Manage all your job applications
+ New Application
Search applications...
All
Active
Archived
ID
COMPANY
TITLE
STATUS
DEADLINE
LINK
#001
Google
Software Engineer
Interview
Nov 15
View
#002
Microsoft
Product Manager
Applied
Nov 20
View
#003
Amazon
Data Scientist
Reviewing
Nov 18
View
#004
Meta
Frontend Developer
Interview
Nov 22
View
#005
Apple
iOS Developer
Rejected
Nov 10
View
#006
Netflix
Backend Engineer
Offer
Nov 25
View
#007
Tesla
ML Engineer
Applied
Nov 28
View
#008
Spotify
Full Stack Developer
Reviewing
Dec 01
View
Showing 1-8 of 24
1
2
3
→


JobTracker
Logout
JD
Dashboard
Applications
Calendar View
Integrations
Settings
Calendar
Deadlines & Interviews
← November 2025 →
Set Reminder
Month
Week
Day
SUN
MON
TUE
WED
THU
FRI
SAT
27
28
29
30
31
1
2
3
4
5
6
Current Date
Today
7
8
9
10
11
12
13
14
15
Google
Interview 2PM
16
17
18
Amazon
Deadline
19
MS Deadline
Apple Follow-up
20
21
22
Meta
Interview 10AM
23
24
25
26
27
28
29
30
1
2
3
4
5
6
7


JobTracker
← Back to Applications
Logout
JD
Dashboard
Applications
Calendar View
Integrations
Settings
G
Software Engineer
Google Inc.
Interview Stage
Update Status
Add Interview
Job Details
Location: Mountain View, CA
Job Type: Full-time
Application Date: October 28, 2025
Deadline: November 15, 2025
Salary Range: $120k - $180k
Experience: 3-5 years
Job Link: careers.google.com/jobs/12345
Timeline & Status
Interview Scheduled
November 15, 2025 at 2:00 PM
Phone Screen Completed
November 5, 2025 - Positive feedback
Application Submitted
October 28, 2025
Application Created
October 25, 2025
Links & Documents
Resume_2025.pdf
Cover_Letter_Google.pdf
Portfolio Website
+ Add Document
Notes
Contacted by recruiter on LinkedIn. Very interested in my experience with React and distributed systems. Prepare system design questions for technical interview.
Interviewer: Sarah Johnson (sarah.j@google.com)


JobTracker
Logout
JD
Dashboard
Applications
Calendar View
Integrations
Settings
Create New Application
Fill in the details for your job application
Step 1 of 1
Company Name *
e.g., Google, Microsoft, Amazon
Job Title *
e.g., Software Engineer, Product Manager
Location
e.g., San Francisco, CA
Job Type
Full-time ▼
Job Link
https://careers.company.com/jobs/...
Application Deadline
MM/DD/YYYY
Salary Range
$80k - $120k
Status *
Select Status ▼
Notes
Add any additional notes, contacts, or important information...
Cancel
Save Application

Testing Plan

Black box tests (API contract and end-user behavior

1. Register new user
   Purpose: validate registration endpoint and DB write.
   Input: POST /api/auth/register JSON { "email":"new@user.test", "username":"newuser", "password":"P@ssw0rd!" }
   Expected output: HTTP 201 Created. Response body empty or { "id": <userId> }. DB row in users with email=new@user.test and non-empty password_hash not equal to raw password.
   Pass: status 201 and DB verification.

2. Register duplicate email
   Purpose: ensure unique-email enforcement.
   Input: same JSON as above, when new@user.test already exists.
   Expected output: HTTP 409 Conflict. Response body { "error":"email exists" }. No new DB row.
   Pass: status 409 and unchanged user count.

3. Login success
   Purpose: validate login and JWT issuance.
   Input: POST /api/auth/login { "email":"new@user.test", "password":"P@ssw0rd!" }
   Expected output: HTTP 200 OK. Body { "jwt":"<token>", "expires_at":"<ISO timestamp>" }. expires_at ≈ now + 15 minutes.
   Pass: status 200, token present, expires_at within 14–16 minutes.

4. Login failure (wrong password)
   Purpose: reject invalid credentials.
   Input: same email, wrong password.
   Expected output: HTTP 401 Unauthorized. Body { "error":"invalid credentials" }.
   Pass: status 401.

5. Access protected resource unauthenticated
   Purpose: enforce auth on /api/apps.
   Input: GET /api/apps with no Authorization header.
   Expected output: HTTP 401 Unauthorized.
   Pass: status 401.

6. CRUD: create application
   Purpose: validate application creation for authenticated user.
   Input: POST /api/apps with valid JWT and body { "company":"ACME","title":"SWE","status":"APPLIED","deadline_at":"2030-01-01T12:00:00Z" }
   Expected output: HTTP 201 Created. Body contains id and stored fields. DB applications row linked to user.
   Pass: status 201 and DB row matches input.

7. CRUD: retrieve applications
   Purpose: list user applications.
   Input: GET /api/apps with valid JWT.
   Expected output: HTTP 200 OK. Body [{ id, company, title, status, deadline_at }].

Pass: status 200 and returned list includes recently created entry.

8. Add reminder
   Purpose: create reminder linked to application.
   Input: POST /api/apps/{id}/reminders { "kind":"INTERVIEW","trigger_at":"2030-01-02T15:00:00Z" } with valid JWT.
   Expected output: HTTP 201 Created. Body with reminder id and trigger_at. DB reminders row with application_id set.
   Pass: status 201 and DB verification.

9. Calendar integration success
   Purpose: server creates Google Calendar event when valid OAuth tokens exist.
   Input: POST /api/apps/{id}/calendar with valid JWT and oauth_tokens present for user. Body { "summary":"Interview","start":"2030-01-02T15:00:00Z","end":"2030-01-02T15:30:00Z" }
   Expected output: HTTP 200 OK. Body { "calendar_event_id":"<id>" }. Calendar event created in user's Google Calendar.
   Pass: status 200 and mock Google API reports event created.

10. Calendar integration fail (no tokens)
    Purpose: handle missing OAuth gracefully.
    Input: same request but user has no OAuth tokens.
    Expected output: HTTP 401 or 400 with { "error":"oauth required" }.
    Pass: appropriate error and no attempt to call Calendar API.

11. Reminder not in past
    Purpose: business rule: no reminder scheduled in the past.
    Input: POST /api/apps/{id}/reminders with trigger_at = now - 1 day.
    Expected output: HTTP 400 Bad Request { "error":"trigger_at must be future" }.
    Pass: status 400 and no DB row.

12. Logout invalidates JWT (stateless or rotation)
    Purpose: logout endpoint behaves as designed.
    Input: POST /api/auth/logout with valid JWT.
    Expected output: HTTP 200 OK. If server tracks tokens, token is rejected afterward. Subsequent GET /api/apps with same token => 401.
    Pass: status 200 then token no longer accepted.

White box tests (internal logic, branches, edge conditions)

13. JWT expiry enforcement
    - Purpose: verify validation logic rejects expired tokens.
    - Input: Generate token with issued_at = now - 16 minutes. Validate token via JwtService.
    - Expected output: validation returns false or throws token-expired error.
    - Pass: validation fails for >15 minutes.

14. Refresh token rotation
    - Purpose: ensure refresh token is invalidated on rotation.

- Input: Use refresh endpoint with current refresh token. Expect new refresh token returned. Try using old refresh token again.
- Expected output: first request returns new refresh token. Second use of old token returns 401.
- Pass: old refresh token rejected.

15. BCrypt password verification
    - Purpose: check hashing and verification paths.
    - Input: store hash via BCrypt. Validate raw password against hash.
    - Expected output: verification returns true. Raw password not stored anywhere in DB.
    - Pass: verification true and DB field password_hash not equal raw password.

16. Reminder duplicate prevention
    - Purpose: scheduler logic avoids resending same reminder.
    - Input: Reminder with trigger_at past and sent_at already set. Scheduler run.
    - Expected output: no new email send, sent_at unchanged.
    - Pass: scheduler skips reminder with non-null sent_at.

17. Reminder concurrency safety
    - Purpose: concurrent scheduler and email scanner modify same application status safely
    - Input: simulate two threads updating applications.status simultaneously
    - Expected output: DB transactions prevent lost update. Final status is consistent and expected based on last committed transaction.
      Pass: no data corruption, proper transaction isolation observed.

18. Email parser regex coverage and false positive check
    - Purpose: ensure parser maps subject/body to status correctly and avoids false matches.
    - Input set:
        - "Interview scheduled for Monday" => INTERVIEW
        - "We regret to inform you" => REJECTED
        - "We are pleased to offer you" => OFFER
        - "Thanks for applying. We will be in touch." => DRAFT or no-change
    - Expected outputs match above mapping.
    - Pass: all mappings correct and ambiguous text returns DRAFT or null mapping.

19. External API retry logic
    - Purpose: validate retry on Google API transient failure
    - Input: Google Calendar API returns 500 on first two calls and 200 on third. Request to create event triggers retry policy.
    - Expected output: service issues 3 attempts with exponential backoff and returns success. Retry counters logged.
    - Pass: event created and total attempts = 3.

20. AES encryption for oauth tokens
    - Purpose: stored tokens are encrypted then decrypted when used.
    - Input: persist access_token plaintext through token storage service. Read back decrypted value.
    - Expected output: DB stores encrypted blob. Decrypted value equals original plaintext.
    - Pass: encrypted field not plaintext and decrypted matches input.

Unit tests (single classes, functions, logic branches)

21. ApplicationService.createApplication validation
    Purpose: verify input validation logic.
    Input: missing company field.
    Expected output: throw ValidationException or return HTTP 400 in controller layer.
    Pass: validation failure triggered.

22. ReminderService.computeNextTrigger
    Purpose: confirm computation of next trigger timestamp for cron-like rules.
    Input: reminder rule "24 hours before interview", interview_at = 2030-01-03T10:00:00Z.
    Expected output: trigger_at = 2030-01-02T10:00:00Z.
    Pass: computed trigger_at equals expected.

23. EmailParser.extractStatus granular checks
    Purpose: unit test regex branches: uppercase, lowercase, punctuation.
    Input cases: "INTERVIEW", "interview", "Interview:", "Your interview is set!"
    Expected output: all map to INTERVIEW.
    Pass: all cases map.

24. GoogleCalendarService.handleExpiredToken
    Purpose: service throws OAuthExpiredException when token expired.
    Input: token with expires_at < now.
    Expected output: exception thrown and no API call attempted.
    Pass: proper exception and no external call.

25. JobFetcher retry backoff timer logic
    Purpose: ensure backoff duration increases on each retry.
    Input: simulate three consecutive failures.
    Expected output: retry delays follow exponential pattern, e.g., 500ms, 1000ms, 2000ms.
    Pass: observed delays match policy.

Regression tests (end-to-end sequences to prevent future breaks)

26. Full happy-path flow
    Purpose: prove core MVP flow remains intact across changes.
    Steps and inputs:
    Register user A
    Login user A and obtain JWT
    Create application for user A with status APPLIED
    Add interview reminder 24 hours before interview_at
    Run scheduler at interview time minus 24 hours
    Expected outputs: reminder.sent_at set, send email API called exactly once, application remains intact.

Pass: all checks pass.

27. CRUD stability after updates
    Purpose: verify updates do not break CRUD.
    Steps: Create app, update title, update status, delete app.
    Expected outputs: each step returns proper HTTP status. After delete, GET returns not found. No dangling reminders.
    Pass: final state has no application or reminders.

28. Calendar integration regression
    Purpose: ensure creation and deletion of calendar events stay consistent after code changes.
    Steps: create calendar event; update app time; update calendar event; delete app and verify calendar event removed if spec requires.
    Expected outputs: calendar event appears and updates reflect new times. If delete should remove event then event removed.
    Pass: external API mocked verifies calls: create, update, delete.

29. Email parsing regression across variants
    Purpose: new email templates do not break parsing.
    Input: set of historical email bodies from production or seed dataset.
    Expected output: statuses mapped as previously expected. No regressions.
    Pass: all historical cases produce same mapping as baseline.

30. Scheduler duplicate-reminder regression
    Purpose: ensure scheduler changes do not introduce duplicate notifications.
    Steps: schedule reminder for T, run scheduler multiple times around T, include restarts.
    Expected output: single notification for reminder, sent_at set only once.
    Pass: exactly one notification call.

Test environment notes (inputs that affect expected outputs)

- Use in-memory DB for unit tests. Expected outputs assume a clean DB at test start.
- For integration tests use H2 or Testcontainers Postgres. If switching to Postgres, timestamps and timezones must be normalized.
- Mock external APIs (Google, Gmail, SendGrid). Tests expect controlled responses. For retry tests, configure mock to fail first N attempts.

Pass/fail criteria summary

- Unit tests: pass when method-level assertions hold.

- White box tests: pass when each branch, exception, and edge condition behaves as specified.
- Black box tests: pass when API responses match expected HTTP statuses and JSON payloads, and DB state matches expectations.
- Regression tests: pass when end-to-end sequences succeed and repeated runs produce identical final state and notification counts.

Deployment

The system consists of three major components:

Backend
• Language: Java 21
• Framework: Spring Boot
• Purpose: Provides the REST API for user authentication, job application management, reminder scheduling, and optional integrations such as Google Calendar and Gmail.

Frontend
• Technology: HTML, CSS, Vanilla JavaScript
• Purpose: Multi-page web interface. Pages interact with backend servlets using AJAX (fetch) for login, registration, calendar updates, and job-tracking features.

Database
• Technology: PostgreSQL
• Purpose: Stores persistent data including users, job applications, reminders, OAuth tokens, and related information.

Deployment Method
• All components run inside Docker containers managed by Docker Compose.

Prerequisites

The following tools must be installed before deployment:

1. Git – used to clone the project repository.
2. Docker – runs the backend, frontend, and database containers.
3. Docker Compose – coordinates the multi-container application.

No installation of Java, Node.js, or PostgreSQL is required on the host system.

Repository Structure

After cloning the repository, the expected folder structure is:

backend – Spring Boot source code
frontend – HTML/CSS source code
docker-compose.yml – Docker configuration
.env – environment variable file created by the user

Environment Configuration

The application uses a .env file to supply configuration variables such as database credentials, API keys, and security settings.

Create a file named .env in the project root. Add the following entries:

POSTGRES_DB=jobtracker

POSTGRES_USER=jobtracker_user

POSTGRES_PASSWORD=your-db-password

SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/jobtracker

SPRING_DATASOURCE_USERNAME=jobtracker_user

SPRING_DATASOURCE_PASSWORD=your-db-password

JWT_SECRET=your-long-random-secret

SPRING_PROFILES_ACTIVE=prod

Optional integrations (these may be left blank):
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
GOOGLE_REDIRECT_URI=
SENDGRID_API_KEY=
EMAIL_FROM_ADDRESS=
ADZUNA_APP_ID=
ADZUNA_APP_KEY=

Notes:
• The hostname “db” refers to the PostgreSQL container defined in docker-compose.yml.
• JWT_SECRET must be a long, securely generated string.

Docker Deployment Instructions

Build and start all services
From the project root, run the following command:

docker compose up --build

This command builds the backend and frontend images, starts the PostgreSQL database container, and launches all services inside a shared Docker network.

Verify the deployment

Check that containers are running:
docker compose ps

Verify backend health:
curl http://localhost:8080/actuator/health
Expected response: status UP.

Verify frontend:
Start the Spring Boot application.
Open a browser and navigate to: http://localhost:8080/
The application's entry page (login or registration) should load successfully.

Stop and restart the application

Stop all services:
docker compose down

Restart in detached mode:
docker compose up -d

If Docker volumes are defined, PostgreSQL data will persist between restarts.

Manual Deployment (Optional)

Although Docker is the preferred method, the following describes a non-Docker deployment.

PostgreSQL Setup
Install PostgreSQL, create the database, create the user, and grant privileges:
CREATE DATABASE jobtracker;
CREATE USER jobtracker_user WITH PASSWORD 'password';
GRANT ALL PRIVILEGES ON DATABASE jobtracker TO jobtracker_user;

Backend Setup
Install Java 21.
Build with Maven: mvn clean package
Run the application: java -jar target/backend.jar --spring.profiles.active=prod

Frontend Setup
Navigate to the src/main/resources/static/ folder inside the project.
The frontend consists of static HTML, CSS, and JavaScript files — no build step is required.

These files are automatically served by Spring Boot when the application is running.
To update the frontend, simply edit the HTML/CSS/JS files and redeploy the project.

Server and Security Settings

Ports
• Backend (Spring Boot): 8080
• Database (PostgreSQL): 3306
• Frontend: Served directly by Spring Boot from /static, so no separate port.

HTTPS
If deployed in production, the entire Spring Boot application (backend + static frontend) can be served behind an HTTPS reverse proxy such as Nginx.

CORS
The backend must allow the frontend origin.

External Service Configuration (Optional)

Google Calendar and Gmail Integration
• Create a Google Cloud Project
• Enable Calendar API and Gmail API
• Create OAuth credentials
• Set GOOGLE_CLIENT_ID, GOOGLE_CLIENT_SECRET, and GOOGLE_REDIRECT_URI in the .env file

SendGrid Integration
• Create a SendGrid account
• Generate an API key
• Set SENDGRID_API_KEY and EMAIL_FROM_ADDRESS

Job Listings API
• Create an account with the chosen job API provider
• Set ADZUNA_APP_ID and ADZUNA_APP_KEY

These are not required for basic deployment.

Smoke Test Checklist

1. Open the frontend in a browser.
2. Register a new user account.
3. Log in using the new user.
4. Create a job application.
5. Verify that the application appears in the dashboard.
6. Update and delete the job application.
7. Add a reminder (if scheduler integration is enabled).
8. Create a Google Calendar event (if integrations are configured).
9. Refresh the page and verify that authentication persists.
10. Review backend logs using docker logs for any errors.