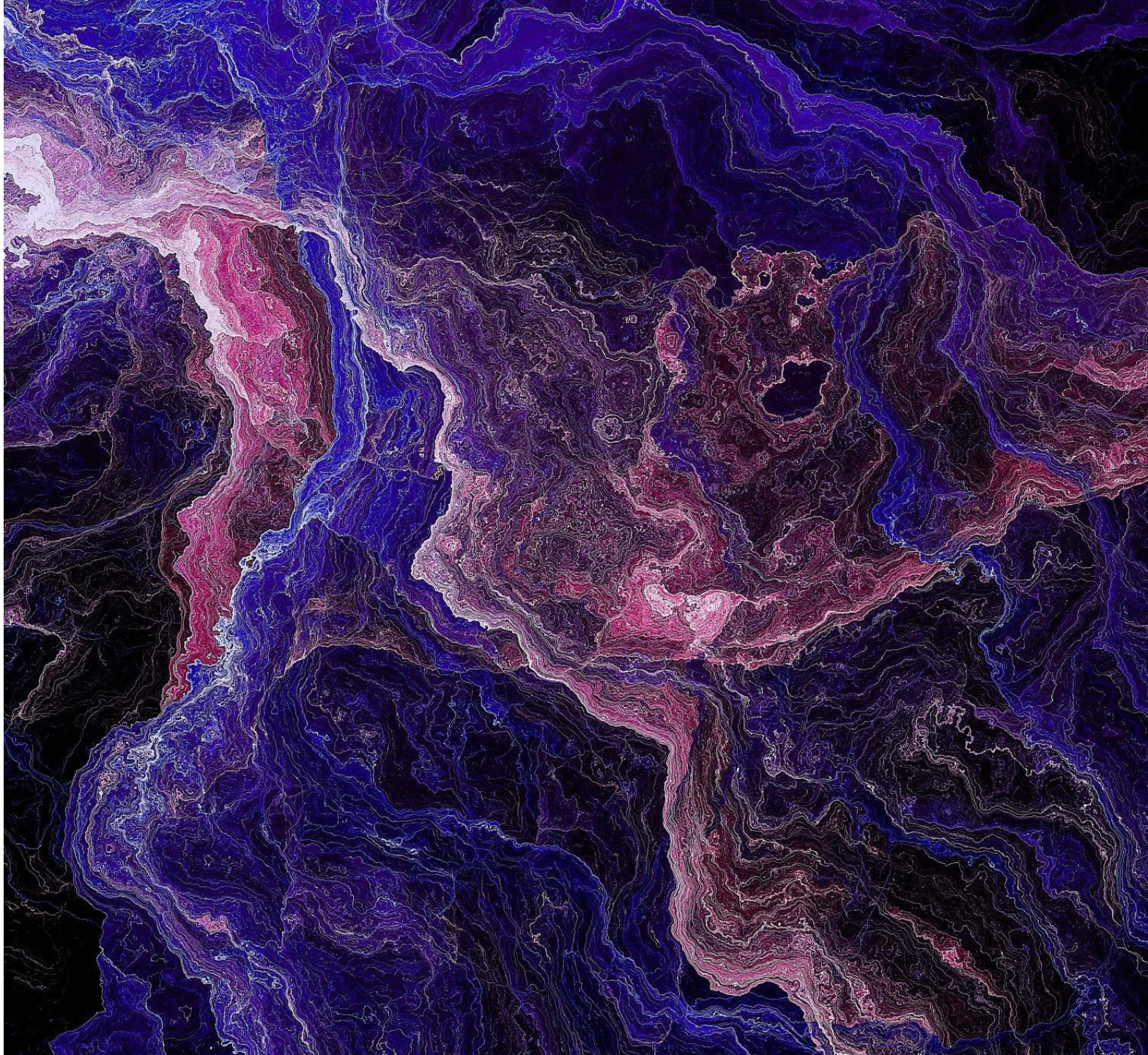
Job Tracker

- A full-stack job application tracker with Kanban workflow and file attachments
- Tech Stack: .NET 8 platform/framework
- Minimal API Style
- EF Core – No raw SQL queries
- React – Modern component-based UI with state management
- JWT – User auth
- AWS S3 – External File Storage



Problem Statement

Why this project exists

I noticed I wanted to keep track of certain companies I have applied to.

This project is for me to track jobs across different companies. Most companies have their own dash board but I needed a centralized place.

Yes maybe LinkedIn can hold all of the applications I have done there but I wanted to be able to customize the information shared. For example, notes for the job.

I also really wanted to challenge myself so I can grow. I really needed something beyond coursework that I could work on and test my skills.

High-Level Architecture

How the system is structured

- Frontend: React (UI + state)
- Backend: .NET 8 Minimal API (business logic)
- Database: SQLite / PostgreSQL (relational data)
- Auth: JWT (stateless authentication)
- Storage: AWS S3 (attachments)

There is a clear separation of concerns

Codebase Map (Important Files)

How the project is organized

Frontend

• `JobAppsPage.tsx` – main page + data loading • `client.ts` – shared API client • `useAuth.ts` – auth state & JWT handling

Backend

• `Program.cs` – middleware & pipeline • `JobAppEndpoints.cs` – feature endpoints • `AppDbContext.cs` – EF Core data access

This slide proves you understand your own codebase.

```
group.MapGet("", async (
    AppDbContext db,
    ClaimsPrincipal user,
    string? q,
    ApplicationStatus? status,
    int page = 1,
    int pageSize = 25) =>
{
    // Authenticated user's ID comes from JWT claims
    var userId = int.Parse(user.FindFirstValue(ClaimTypes.NameIdentifier)!);

    // Guardrails for paging input
    if (page < 1) page = 1;
    if (pageSize < 1) pageSize = 25;
    if (pageSize > 100) pageSize = 100;

    // Ownership enforcement: only fetch apps belonging to the current user
    IQueryable<JobApplication> query = db.JobApplications
        .AsNoTracking()
        .Where(x => x.UserId == userId);

    // Text search across Company, RoleTitle, Notes (LIKE query)
    if (!string.IsNullOrWhiteSpace(q))
    {
        var term = $"%{q.Trim()}%";

        query = query.Where(x =>
            (x.Company != null && EF.Functions.Like(x.Company, term)) ||
            (x.RoleTitle != null && EF.Functions.Like(x.RoleTitle, term)) ||
            (x.Notes != null && EF.Functions.Like(x.Notes, term))
        );
    }

    // Optional status filter (maps to Kanban lanes)
    if (status is not null)
        query = query.Where(x => x.Status == status);
```

Core Code Block #1 (Backend Logic)

- This endpoint lists job applications for the logged-in user.
- I get the user ID from the JWT claims using NameIdentifier.
- Ownership is enforced at the query level with `Where(UserId == userId)` so users can never access someone else's data.
- I use EF Core with `IQueryable` so filters like search, status, and paging compose cleanly before the SQL is executed.
- `AsNoTracking()` improves read performance since this is a read-only list.
- Paging is implemented with `Skip` and `Take` to prevent returning large result sets.
- I return DTOs instead of entities so the API contract remains stable and persistence details are not exposed.

Core Code Block #2 (Frontend Logic)

- Handles Kanban drag-and-drop status changes
- Uses optimistic UI updates for immediate feedback
- Sends minimal PATCH request to backend API
- Rolls back UI state if the request fails
- Keeps frontend state consistent with backend data

```tsx
/**
 * moveToLane (Kanban move -> Backend status update)
 * Interview talking point:
 * - Optimistic UI update for snappy UX
 * - PATCH minimal payload: { status }
 * - Rollback if request fails (preserves data integrity)
 */
async function moveToLane(jobId: number, nextLane: BoardLane) {
  const job = items.find((x) => x.id === jobId);
  if (!job) return;

  // Optimistic UI update: user sees the move instantly
  setItems((prev) => prev.map((x) => (x.id === jobId ? { ...x, status: nextLane } : x)));

  try {
    // Minimal PATCH payload: only update what changed
    await updateJobApp(jobId, { status: nextLane });
    toast.success(`Moved to ${laneLabel(nextLane)}.`);
  } catch (err: unknown) {
    // Debug info to help diagnose enum mismatches or auth issues
    console.error("MOVE FAILED", err);

    const maybeAxiosErr = err as {
      response?: { status?: number; data?: unknown };
      message?: string;
    };

    console.error("MOVE FAILED status:", maybeAxiosErr?.response?.status);
    console.error("MOVE FAILED data:", maybeAxiosErr?.response?.data);

    // Rollback: restore the original job state if the API call fails
    requestAnimationFrame(() => {
      setItems((prev) => prev.map((x) => (x.id === jobId ? job : x)));
    });

    toast.error("Move failed.");
  }
}
```

Data and Security Design

- JWT-based authentication on every request
- Authorization enforced at the data access layer
- EF Core queries scoped by authenticated user
- Attachments stored in AWS S3 with metadata in the database
- Defense-in-depth across frontend and backend

Tradeoffs

Minimal APIs for explicit routing and reduced boilerplate

Benefit: Faster development, fewer moving parts, clear endpoint mapping.
Tradeoff: As the app grows, endpoints can become harder to organize and discover compared to a more opinionated controller structure.
Mitigation: Use feature-based folders, endpoint grouping (route groups), consistent naming, and shared helpers (DTO mapping, validation, auth checks).

Tradeoffs

EF Core for strongly typed data access and provider flexibility

Benefit: Strong typing, faster iteration, migrations, relationships, and provider abstraction (SQLite locally, PostgreSQL in production).
Tradeoff: ORM abstractions can hide performance issues (N+1 queries, over-fetching, unexpected SQL generation).
Mitigation: Use projection to DTOs, carefully shape queries, avoid unnecessary includes, inspect generated SQL when needed, and add indexes for hot paths.

Tradeoffs

Optimistic UI updates balanced with rollback logic

Benefit: UI feels instant and responsive; users see changes immediately without waiting on round trips.

Tradeoff: If a request fails, the UI can temporarily show incorrect state and needs correction, which adds complexity.

Mitigation: Keep a previous-state snapshot, show clear error toasts, re-fetch on failure, and scope optimistic updates to low-risk actions.

Tradeoffs

Separate attachment handling for clean boundaries

Benefit: Clear separation between core domain data and file storage concerns; easier scaling, security, and future storage changes (S3, presigned URLs).
Tradeoff: More endpoints, more coordination (metadata in database + file in storage), and more failure cases (uploaded file succeeds but DB write fails, or vice versa).
Mitigation: Treat attachments as a separate workflow with status/validation, enforce size/type limits, use predictable storage keys, and handle cleanup or retries when one step fails.

Tradeoffs

Simplicity prioritized where appropriate

Benefit: Lower cognitive load, faster shipping, easier maintenance, and fewer bug surfaces.

Tradeoff: Some decisions may not be “max scalable” on day one, and future advanced requirements might require refactoring.

Mitigation: Keep clean boundaries and interfaces so upgrades are contained (service layer, DTOs, modular endpoints), and refactor when real requirements appear.

Challenges and What I Learned

KEEPING UI STATE SHAREABLE AND REFRESH SAFE

PREVENTING STALE API RESPONSES DURING RAPID CHANGES

ENSURING FRONTEND AND BACKEND ENUMS STAY ALIGNED

BALANCING RICH INTERACTIONS WITH PREDICTABLE BEHAVIOR

Wrapping up & Demo

- Live demo of core workflows: https://jobtracker-b0k3.onrender.com
- Quick codebase orientation
- Key takeaways from the project
- Open to questions and deep dives