Contents

Job Application Tracker - Comprehensive Project Report

Project Name: Job Application Tracker (Full Stack)
Developer: Kaysarul Anas **Technology Stack:** Django REST Framework + React (Vite)
Report Date: February 8, 2026
Project Status: Active Development

Executive Summary

The Job Application Tracker is a full-stack web application designed to help job seekers organize and manage their job applications efficiently. The system provides secure user authentication, comprehensive job tracking capabilities, file management for resumes and cover letters, and a modern, responsive user interface. Built with industry-standard technologies, the application demonstrates proficiency in both backend API development and modern frontend frameworks.

Key Achievements

- Secure JWT-based authentication with Google OAuth integration
- RESTful API with Django REST Framework
- Modern React frontend with Tailwind CSS v4
- User-specific data isolation and security

- File upload and management system
- Responsive dashboard with multiple view modes
- Production-ready deployment configuration

Table of Contents

---

1. Problem Statement

The Challenge

Job seekers often struggle with:

- **Disorganization**: Tracking multiple applications across different companies and platforms
- **Lost Information**: Forgetting which resume version was sent to which company
- **Missed Follow-ups**: Losing track of application statuses and interview dates
- **Scattered Data**: Job descriptions, URLs, and notes stored in various locations
- **No Analytics**: Unable to see patterns in application success rates

Target Users

- Recent graduates entering the job market
- Career changers managing multiple applications
- Active job seekers applying to 10+ positions simultaneously
- Professionals who want to maintain organized job search records

---

2. Solution Overview

Core Features

Secure Authentication

- Email/password registration and login

- Google OAuth integration for quick access
- JWT token-based session management
- Automatic token refresh mechanism

Application Tracking

- Track company name, position, and application status
- Record application dates and deadlines
- Store job post URLs for easy reference
- Save job requirements for interview preparation
- Add personal notes and observations

File Management

- Upload multiple files per application (resumes, cover letters, portfolios)
- Track which version of documents was sent to each company
- Secure file storage with user-specific access control

Dashboard & Analytics

- Visual statistics showing application status breakdown
- Board view (Kanban-style) for drag-and-drop organization
- List view for detailed information display
- Search and filter capabilities
- Demo mode for exploring features without login

User Feedback System

- Submit reviews and ratings
- Public testimonials for homepage
- Moderation system for review approval

---

3. System Architecture

High-Level Architecture

```
graph TB
    subgraph "Frontend Layer"
        A[React Application<br/>Vite + Tailwind CSS v4]
        B[React Router<br/>Client-side Navigation]
        C[Axios Client<br/>API Communication]
        D[Auth Context<br/>State Management]
    end

    subgraph "Backend Layer"
        E[Django REST Framework<br/>API Server]
        F[SimpleJWT<br/>Authentication]
        G[Django Allauth<br/>Social Auth]
```

```
    H[CORS Middleware<br/>Cross-Origin Requests]
end

subgraph "Data Layer"
    I[SQLite Database<br/>Development]
    J[PostgreSQL<br/>Production]
    K[File Storage<br/>Application Files]
end

A --> C
B --> A
D --> A
C --> H
H --> E
E --> F
E --> G
E --> I
E --> J
E --> K

style A fill:#61dafb,stroke:#333,stroke-width:2px
style E fill:#092e20,stroke:#333,stroke-width:2px,color:#fff
style I fill:#003b57,stroke:#333,stroke-width:2px,color:#fff
```

Technology Stack Overview

Layer	Technology	Purpose
Frontend	React 19.2	UI component library
	Vite 7.2	Build tool and dev server
	Tailwind CSS v4	Utility-first CSS framework
	React Router v7	Client-side routing
	Axios	HTTP client for API calls
Backend	Django 5.2	Web framework
	Django REST Framework 3.16	API development
	SimpleJWT 5.5	JWT authentication
	Django Allauth 65.14	Social authentication
Database	SQLite	Development database
	PostgreSQL	Production database
DevOps	Gunicorn	WSGI HTTP server
	WhiteNoise	Static file serving
	Docker	Containerization

Request Flow Diagram

```
sequenceDiagram
    participant User
    participant React
```

```
    participant Axios
    participant Django
    participant Database

    User->>React: Login with credentials
    React->>Axios: POST /api/auth/token/
    Axios->>Django: Forward request
    Django->>Database: Validate credentials
    Database-->>Django: User data
    Django-->>Axios: JWT tokens (access + refresh)
    Axios-->>React: Store tokens in localStorage
    React->>Axios: GET /api/applications/
    Axios->>Axios: Attach Bearer token
    Axios->>Django: Authenticated request
    Django->>Django: Verify JWT token
    Django->>Database: Query user's applications
    Database-->>Django: Application data
    Django-->>Axios: JSON response
    Axios-->>React: Update UI with data
    React-->>User: Display dashboard
```

---

4. Backend Development

4.1 Technology Stack

The backend is built with **Django 5.2** and **Django REST Framework 3.16**, providing a robust, scalable API architecture.

Core Dependencies

```
# requirements.txt
Django==5.2.11                      # Web framework
djangorestframework==3.16.1         # REST API toolkit
djangorestframework_simplejwt==5.5.1 # JWT authentication
django-allauth==65.14.0             # Social authentication
django-cors-headers==4.9.0          # CORS handling
dj-rest-auth==7.0.2                 # Auth endpoints
gunicorn==23.0.0                    # Production server
psycopg2-binary==2.9.10            # PostgreSQL adapter
dj-database-url==2.3.0             # Database URL parsing
whitenoise==6.9.0                   # Static file serving
python-dotenv==1.2.1                # Environment variables
```

4.2 Project Structure

```
backend/
  backend/                # Project configuration
     settings.py         # Django settings
```

```
    urls.py             # URL routing
    wsgi.py             # WSGI application
  applications/          # Main app
    models.py           # Database models
    serializers.py      # DRF serializers
    views.py            # API views
    admin.py            # Admin configuration
    migrations/         # Database migrations
  manage.py              # Django management script
  requirements.txt       # Python dependencies
  Dockerfile            # Container configuration
  db.sqlite3           # Development database
```

4.3 Django Settings Configuration

Key configurations in `settings.py`:

Installed Apps

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'whitenoise.runserver_nostatic',
    'django.contrib.staticfiles',
    'rest_framework.authtoken',
    'rest_framework',
    'rest_framework_simplejwt',
    'corsheaders',
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    'dj_rest_auth',
    'dj_rest_auth.registration',
    'applications',  # Custom app
]
```

REST Framework Configuration

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),
```

```
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

JWT Configuration

```
from datetime import timedelta

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'AUTH_HEADER_TYPES': ('Bearer',),
}
```

CORS Configuration

```
CORS_ALLOWED_ORIGINS = os.getenv(
    'CORS_ALLOWED_ORIGINS',
    'http://localhost:5173'
).split(',')
CORS_ALLOW_ALL_ORIGINS = DEBUG  # Only in development
```

4.4 Database Models

Application Model The core model for tracking job applications:

```
from django.db import models
from django.contrib.auth import get_user_model

User = get_user_model()

class Application(models.Model):
    # Status choices
    APPLIED = 'Applied'
    INTERVIEW = 'Interview'
    OFFER = 'Offer'
    REJECTED = 'Rejected'

    STATUS_CHOICES = [
        (APPLIED, 'Applied'),
        (INTERVIEW, 'Interview'),
        (OFFER, 'Offer'),
        (REJECTED, 'Rejected'),
    ]

    # Relationships
    user = models.ForeignKey(
        User,
```

```
        on_delete=models.CASCADE,
        related_name="applications"
    )

    # Core fields
    company_name = models.CharField(max_length=255)
    position_title = models.CharField(max_length=255)
    notes = models.TextField(blank=True, null=True)

    # Job details
    job_post_url = models.URLField(blank=True, null=True)
    job_requirements = models.TextField(blank=True, null=True)

    # Status tracking
    status = models.CharField(
        max_length=50,
        choices=STATUS_CHOICES,
        default=APPLIED
    )
    applied_at = models.DateField(blank=True, null=True)

    # Timestamps
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.company_name} - {self.position_title}"
```

Design Decisions:

- User foreign key ensures data isolation
- Status choices provide consistent data
- Timestamps track creation and updates
- Optional fields allow flexibility

ApplicationFile Model Handles multiple file uploads per application:

```
class ApplicationFile(models.Model):
    application = models.ForeignKey(
        Application,
        on_delete=models.CASCADE,
        related_name="files"
    )

    file = models.FileField(upload_to="application_files/")
    file_type = models.CharField(max_length=50)
    original_filename = models.CharField(max_length=255)

    created_at = models.DateTimeField(auto_now_add=True)
```

```
    def __str__(self):
        return self.original_filename
```

Features:

- One-to-many relationship with Application
- Cascade delete (files deleted with application)
- Original filename preservation
- File type tracking (resume, cover letter, etc.)

Review Model User feedback and testimonials:

```
class Review(models.Model):
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="reviews"
    )
    rating = models.IntegerField(default=5)  # 1-5 stars
    comment = models.TextField()
    is_public = models.BooleanField(default=False)  # Moderation

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"Review by {self.user.username} - {self.rating} stars"
```

Features:

- Moderation system (is_public flag)
- Star rating system (1-5)
- User attribution

4.5 Database Relationships

```
erDiagram
    USER ||--o{ APPLICATION : creates
    USER ||--o{ REVIEW : writes
    APPLICATION ||--o{ APPLICATION_FILE : contains

    USER {
        int id PK
        string username
        string email
        string password
        datetime date_joined
    }
```

```
    APPLICATION {
        int id PK
        int user_id FK
        string company_name
        string position_title
        text notes
        string job_post_url
        text job_requirements
        string status
        date applied_at
        datetime created_at
        datetime updated_at
    }

    APPLICATION_FILE {
        int id PK
        int application_id FK
        file file
        string file_type
        string original_filename
        datetime created_at
    }

    REVIEW {
        int id PK
        int user_id FK
        int rating
        text comment
        boolean is_public
        datetime created_at
        datetime updated_at
    }
```

4.6 API Serializers

Serializers convert Django models to JSON and validate incoming data:

ApplicationSerializer

```python
from rest_framework import serializers
from .models import Application, ApplicationFile

class ApplicationFileSerializer(serializers.ModelSerializer):
    class Meta:
        model = ApplicationFile
        fields = ['id', 'file', 'file_type', 'original_filename', 'created_at']

class ApplicationSerializer(serializers.ModelSerializer):
```

```
    files = ApplicationFileSerializer(many=True, read_only=True)

    class Meta:
        model = Application
        fields = '__all__'
        read_only_fields = ['user']
```

Key Features:

- Nested serialization (files within application)
- Read-only user field (set automatically)
- Automatic validation

ReviewSerializer

```
class ReviewSerializer(serializers.ModelSerializer):
    username = serializers.CharField(source='user.username', read_only=True)
    display_name = serializers.CharField(source='user.first_name', read_only=True)

    class Meta:
        model = Review
        fields = ['id', 'username', 'display_name', 'rating', 'comment',
                  'is_public', 'created_at']
        read_only_fields = ['user', 'is_public']
```

Features:

- Computed fields (username, display_name)
- Moderation control (is_public read-only)

4.7 API Views

ViewSets provide CRUD operations with minimal code:

ApplicationViewSet

```
from rest_framework import viewsets, permissions

class ApplicationViewSet(viewsets.ModelViewSet):
    serializer_class = ApplicationSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        # Only return applications belonging to current user
        return self.request.user.applications.all()

    def perform_create(self, serializer):
        # Automatically set user to current logged-in user
        serializer.save(user=self.request.user)
```

Security Features:

- Authentication required
- User-specific data filtering
- Automatic user assignment
- No cross-user data access

ApplicationFileViewSet

```
class ApplicationFileViewSet(viewsets.ModelViewSet):
    serializer_class = ApplicationFileSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        # Only return files from current user's applications
        return ApplicationFile.objects.filter(
            application__user=self.request.user
        )
```

Security:

- Double-layer filtering (application → user)
- Prevents unauthorized file access

ReviewViewSet

```
class ReviewViewSet(viewsets.ModelViewSet):
    serializer_class = ReviewSerializer

    def get_permissions(self):
        if self.action == 'list':
            return [permissions.AllowAny()]
        return [permissions.IsAuthenticated()]

    def get_queryset(self):
        if self.action == 'list':
            return Review.objects.filter(is_public=True).order_by('-created_at')
        return Review.objects.filter(user=self.request.user)

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

Features:

- Public reviews visible to all
- Private reviews only to owner
- Automatic moderation (is_public=False by default)

---

5. Frontend Development

5.1 Technology Stack

The frontend leverages modern React ecosystem tools for optimal developer experience and performance.

Core Technologies

```
{
  "dependencies": {
    "react": "^19.2.0", // UI library
    "react-dom": "^19.2.0", // DOM rendering
    "react-router-dom": "^7.13.0", // Routing
    "axios": "^1.13.4", // HTTP client
    "@react-oauth/google": "^0.13.4", // Google OAuth
    "lucide-react": "^0.563.0" // Icon library
  },
  "devDependencies": {
    "vite": "^7.2.4", // Build tool
    "tailwindcss": "^4.1.18", // CSS framework
    "@tailwindcss/vite": "^4.1.18", // Tailwind Vite plugin
    "@vitejs/plugin-react": "^5.1.1", // React plugin
    "eslint": "^9.39.1" // Linting
  }
}
```

5.2 Project Structure

```
frontend/
   src/
       api/
           axios.js              # Axios instance & interceptors
       components/
           AuthLayout.jsx        # Auth page wrapper
           Navbar.jsx            # Navigation bar
           Footer.jsx            # Footer component
           LandingHero.jsx       # Homepage hero
           Features.jsx          # Feature showcase
           Testimonials.jsx      # User reviews
           dashcomp/             # Dashboard components
               DashboardHeader.jsx
               StatsSection.jsx
               BoardView.jsx
               ListView.jsx
               ApplicationCard.jsx
               SearchControls.jsx
               ReviewForm.jsx
       context/
```

```
        AuthContext.jsx       # Authentication state
    pages/
        HomePage.jsx          # Landing page
        LoginPage.jsx         # Login form
        SignupPage.jsx        # Registration form
        Dashboard/
            DashBoard.jsx     # Main dashboard
        NotFoundPage.jsx      # 404 page
    data/
        demoData.js           # Demo mode data
    App.jsx                   # Main app component
    main.jsx                  # Entry point
    index.css                 # Global styles
  public/                     # Static assets
  index.html                  # HTML template
  vite.config.js              # Vite configuration
  package.json                # Dependencies
```

5.3 Routing Architecture

```jsx
// App.jsx
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";

const PrivateRoute = ({ children }) => {
  const { user, loading } = useAuth();

  if (loading) return <Loader variant="full-screen" />;
  return user ? children : <Navigate to="/login" />;
};

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/signup" element={<SignupPage />} />
        <Route
          path="/dashboard"
          element={
            <PrivateRoute>
              <DashBoard />
            </PrivateRoute>
```

```
          }
        />
        <Route path="/demo" element={<DashBoard isDemo={true} />} />
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </Router>
  );
}
```

Features:

- Protected routes with authentication check
- Loading states during auth verification
- Demo mode for unauthenticated exploration
- 404 fallback route

5.4 API Integration

Axios Configuration

```
// src/api/axios.js
import axios from "axios";

const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL || "http://127.0.0.1:8000/",
  headers: {
    "Content-Type": "application/json",
  },
});

// Request interceptor: Attach JWT token
api.interceptors.request.use(
  (config) => {
    // Skip auth for login/register endpoints
    const authEndpoints = ["/api/auth/token/", "/api/auth/registration/"];
    const isAuthEndpoint = authEndpoints.some((endpoint) =>
      config.url.includes(endpoint)
    );

    if (!isAuthEndpoint) {
      const token = localStorage.getItem("access_token");
      if (token) {
        config.headers.Authorization = `Bearer ${token}`;
      }
    }

    return config;
  },
  (error) => Promise.reject(error)
```

```
);

// Response interceptor: Handle token refresh
api.interceptors.response.use(
  (response) => response,
  async (error) => {
    const originalRequest = error.config;

    // If 401 and not already retried, try to refresh token
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;

      const refreshToken = localStorage.getItem("refresh_token");
      if (refreshToken) {
        try {
          const response = await axios.post(
            `${api.defaults.baseURL}api/auth/token/refresh/`,
            { refresh: refreshToken }
          );

          localStorage.setItem("access_token", response.data.access);
          originalRequest.headers.Authorization = `Bearer ${response.data.access}`;

          return api(originalRequest);
        } catch (refreshError) {
          // Refresh failed, logout user
          localStorage.clear();
          window.location.href = "/login";
        }
      }
    }

    return Promise.reject(error);
  }
);

export default api;
```

Key Features:

- Automatic token attachment
- Token refresh on 401 errors
- Environment-based API URL
- Graceful logout on auth failure

5.5 State Management with Context API

AuthContext

```jsx
// src/context/AuthContext.jsx
import { createContext, useContext, useState, useEffect } from "react";
import api from "../api/axios";

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  // Check for existing session on mount
  useEffect(() => {
    const initAuth = async () => {
      const token = localStorage.getItem("access_token");
      if (token) {
        try {
          const response = await api.get("/api/auth/user/");
          setUser(response.data);
        } catch (error) {
          localStorage.clear();
        }
      }
      setLoading(false);
    };

    initAuth();
  }, []);

  const login = async (credentials) => {
    const response = await api.post("/api/auth/token/", credentials);
    const { access, refresh } = response.data;

    localStorage.setItem("access_token", access);
    localStorage.setItem("refresh_token", refresh);

    // Fetch user profile
    const userResponse = await api.get("/api/auth/user/");
    setUser(userResponse.data);
  };

  const logout = () => {
    localStorage.clear();
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, loading, login, logout }}>
      {children}
```

```
    </AuthContext.Provider>
  );
};

export const useAuth = () => useContext(AuthContext);
```

Features:

- Persistent sessions across page refreshes
- Automatic user profile fetching
- Centralized authentication state
- Clean logout functionality

---

6. UI/UX Design

6.1 Design Philosophy

The application follows modern web design principles with a focus on:

1. **Minimalism**: Clean, uncluttered interfaces
2. **Responsiveness**: Mobile-first design approach
3. **Accessibility**: Semantic HTML and ARIA labels
4. **Performance**: Optimized assets and lazy loading
5. **Consistency**: Unified color scheme and typography

6.2 Design System

Color Palette The application uses Tailwind CSS v4 with custom theme configuration:

```
/* src/index.css */
@import "tailwindcss";

@theme {
  --font-sans: Inter, -apple-system, BlinkMacSystemFont, "Segoe UI", sans-serif;
}

body {
  @apply font-sans;
}
```

Color Usage:

- **Primary**: Blue tones for CTAs and interactive elements
- **Success**: Green for positive actions (Offer status)
- **Warning**: Yellow/Orange for pending states (Interview)
- **Danger**: Red for rejections and destructive actions
- **Neutral**: Gray scale for text and backgrounds

Typography

- **Font Family**: Inter (Google Fonts) for modern, readable text

- **Headings**: Bold weights (600-700) for hierarchy
- **Body Text**: Regular weight (400) for readability
- **Code**: Monospace for technical content

6.3 Component Design

Landing Page

```
graph TD
    A[Navbar] --> B[Hero Section]
    B --> C[Features Section]
    C --> D[Testimonials]
    D --> E[CTA Section]
    E --> F[Footer]

    style A fill:#3b82f6,color:#fff
    style B fill:#10b981,color:#fff
    style C fill:#8b5cf6,color:#fff
```

Key Elements:

- **Hero**: Eye-catching headline with demo CTA
- **Features**: Grid layout showcasing core capabilities
- **Testimonials**: User reviews with star ratings
- **CTA**: Strong call-to-action for signup

Dashboard Layout

```
        Dashboard Header
[Logo] [Search] [View Toggle] [User]


Stats Section (4 cards)
[Applied] [Interview] [Offer] [Reject]



Board View / List View
(Application Cards)
```

Responsive Design

- **Desktop (1024px)**: Full sidebar, multi-column layouts
- **Tablet (768px-1023px)**: Collapsed sidebar, 2-column grids
- **Mobile (<768px)**: Stacked layout, hamburger menu

6.4 User Flows

Registration Flow

```
flowchart LR
    A[Visit Homepage] --> B{Authenticated?}
    B -->|No| C[Click Get Started]
    C --> D[Signup Page]
    D --> E{Choose Method}
    E -->|Email| F[Fill Form]
    E -->|Google| G[OAuth Popup]
    F --> H[Submit]
    G --> H
    H --> I[Redirect to Dashboard]
    B -->|Yes| I
```

Application Management Flow

```
flowchart TD
    A[Dashboard] --> B[View Applications]
    B --> C{Action}
    C -->|Add New| D[Create Form]
    C -->|Edit| E[Update Form]
    C -->|Delete| F[Confirm Delete]
    C -->|View Details| G[Detail Modal]
    D --> H[Save]
    E --> H
    F --> H
    H --> B
```

---

7. Authentication & Security

7.1 Authentication Architecture

The application implements a dual-authentication system:

1. **Email/Password**: Traditional credentials-based auth
2. **Google OAuth**: Social authentication for convenience

Authentication Flow

```
sequenceDiagram
    participant User
    participant Frontend
    participant Backend
    participant Google
    participant Database

    alt Email/Password Login
```

```
        User->>Frontend: Enter credentials
        Frontend->>Backend: POST /api/auth/token/
        Backend->>Database: Validate user
        Database-->>Backend: User data
        Backend-->>Frontend: JWT tokens
    else Google OAuth Login
        User->>Frontend: Click Google Login
        Frontend->>Google: OAuth request
        Google-->>User: Consent screen
        User->>Google: Approve
        Google-->>Frontend: Authorization code
        Frontend->>Backend: POST /api/auth/google/
        Backend->>Google: Verify token
        Google-->>Backend: User info
        Backend->>Database: Create/update user
        Database-->>Backend: User data
        Backend-->>Frontend: JWT tokens
    end

    Frontend->>Frontend: Store tokens
    Frontend->>Backend: GET /api/auth/user/
    Backend-->>Frontend: User profile
    Frontend->>User: Redirect to dashboard
```

7.2 Security Measures

Token Management Access Token:

- Lifetime: 60 minutes
- Storage: localStorage
- Usage: Attached to all API requests

Refresh Token:

- Lifetime: 1 day
- Storage: localStorage
- Usage: Obtain new access token when expired

[!WARNING] > **Production Consideration**: For enhanced security, consider using httpOnly cookies instead of localStorage to prevent XSS attacks.

CORS Configuration

```python
# Development
CORS_ALLOW_ALL_ORIGINS = True

# Production
CORS_ALLOWED_ORIGINS = [
    'https://yourdomain.com',
```

```
    'https://www.yourdomain.com'
]
```

Data Isolation Every API endpoint implements user-based filtering:

```
def get_queryset(self):
    return self.request.user.applications.all()
```

This ensures users can only access their own data.

7.3 Authentication Challenges & Solutions

During development, several authentication issues were encountered and resolved:

Challenge 1: Registration Failures **Problem**: Backend rejected valid registration data
Solution: Added `password2` field for confirmation

Challenge 2: Google OAuth 500 Errors **Problem**: Django Sites framework misconfiguration
Solution: Corrected `SITE_ID` and linked Social Application

Challenge 3: 401 Unauthorized Loop **Problem**: Axios interceptor attached tokens to login requests
Solution: Excluded auth endpoints from token attachment

Challenge 4: Token Key Mismatch **Problem**: Backend sent `key`, frontend expected `access`
Solution: Implemented dual-format token detection

---

8. Database Design

8.1 Schema Overview

The database consists of 4 main tables:

1. **User** (Django built-in)
2. **Application** (Custom)
3. **ApplicationFile** (Custom)
4. **Review** (Custom)

8.2 Data Relationships

```
graph LR
    A[User] -->|1:N| B[Application]
    A -->|1:N| C[Review]
    B -->|1:N| D[ApplicationFile]

    style A fill:#3b82f6,color:#fff
    style B fill:#10b981,color:#fff
    style C fill:#f59e0b,color:#fff
    style D fill:#8b5cf6,color:#fff
```

8.3 Database Migrations

Django's migration system tracks schema changes:

```
# Create migrations
python manage.py makemigrations

# Apply migrations
python manage.py migrate

# View migration status
python manage.py showmigrations
```

8.4 Sample Data

For development and testing, sample data can be loaded:

```
# Create demo applications
Application.objects.create(
    user=user,
    company_name="Google",
    position_title="Software Engineer",
    status="Interview",
    applied_at="2026-01-15"
)
```

---

9. API Documentation

9.1 Base URL

- **Development**: `http://127.0.0.1:8000/`
- **Production**: `https://your-domain.com/`

9.2 Authentication Endpoints

POST /api/auth/token/ Obtain JWT access and refresh tokens.

Request:

```
{
  "email": "user@example.com",
  "password": "securepassword"
}
```

Response:

```
{
  "access": "eyJ0eXAiOiJKV1QiLCJhbGc...",
  "refresh": "eyJ0eXAiOiJKV1QiLCJhbGc..."
}
```

POST /api/auth/token/refresh/ Refresh access token using refresh token.

Request:

```
{
  "refresh": "eyJ0eXAiOiJKV1QiLCJhbGc..."
}
```

Response:

```
{
  "access": "eyJ0eXAiOiJKV1QiLCJhbGc..."
}
```

POST /api/auth/registration/ Register new user account.

Request:

```
{
  "email": "newuser@example.com",
  "password1": "securepassword",
  "password2": "securepassword"
}
```

GET /api/auth/user/ Get current user profile (requires authentication).

Response:

```
{
  "id": 1,
  "username": "user@example.com",
  "email": "user@example.com",
  "first_name": "John",
  "last_name": "Doe"
}
```

9.3 Application Endpoints

GET /api/applications/ List all applications for authenticated user.

Response:

```
[
  {
    "id": 1,
    "user": 1,
    "company_name": "Google",
    "position_title": "Software Engineer",
    "notes": "Great company culture",
    "job_post_url": "https://careers.google.com/...",
    "job_requirements": "Python, Django, React",
    "status": "Interview",
    "applied_at": "2026-01-15",
```

```
    "created_at": "2026-01-15T10:30:00Z",
    "updated_at": "2026-01-20T14:45:00Z",
    "files": [
      {
        "id": 1,
        "file": "/media/application_files/resume.pdf",
        "file_type": "resume",
        "original_filename": "john_doe_resume.pdf",
        "created_at": "2026-01-15T10:35:00Z"
      }
    ]
  }
]
```


POST /api/applications/ Create new application.

Request:


```
{
  "company_name": "Microsoft",
  "position_title": "Frontend Developer",
  "status": "Applied",
  "applied_at": "2026-02-08",
  "notes": "Applied through LinkedIn",
  "job_post_url": "https://careers.microsoft.com/..."
}
```


GET /api/applications/{id}/ Retrieve specific application.

PUT /api/applications/{id}/ Update application.

DELETE /api/applications/{id}/ Delete application.

9.4 File Endpoints

GET /api/files/ List all files for user's applications.

POST /api/files/ Upload file for an application.

Request (multipart/form-data):

```
application: 1
file: [binary data]
file_type: "resume"
original_filename: "resume_v2.pdf"
```

9.5 Review Endpoints

GET /api/reviews/ List public reviews (no auth required).

POST /api/reviews/ Submit new review (requires authentication).

Request:

```
{
  "rating": 5,
  "comment": "This app helped me organize my job search perfectly!"
}
```

---

10. Deployment & DevOps

10.1 Development Setup

Backend Setup

```
cd backend
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

Frontend Setup

```
cd frontend
npm install
npm run dev
```

10.2 Environment Variables

Backend (.env)

```
SECRET_KEY=your-secret-key
DEBUG=True
ALLOWED_HOSTS=localhost,127.0.0.1
CORS_ALLOWED_ORIGINS=http://localhost:5173
DATABASE_URL=sqlite:///db.sqlite3
```

Frontend (.env)

```
VITE_API_URL=http://127.0.0.1:8000/
VITE_GOOGLE_CLIENT_ID=your-google-client-id
```

10.3 Production Deployment

Docker Configuration

```
# Dockerfile
FROM python:3.14-slim
```

```
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

RUN python manage.py collectstatic --noinput

CMD ["gunicorn", "backend.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Production Checklist

- ☐ Set `DEBUG=False`
- ☐ Configure proper `ALLOWED_HOSTS`
- ☐ Use PostgreSQL instead of SQLite
- ☐ Set up static file serving (WhiteNoise)
- ☐ Configure HTTPS
- ☐ Set secure CORS origins
- ☐ Use environment variables for secrets
- ☐ Set up database backups
- ☐ Configure logging
- ☐ Enable CSRF protection

---

11. Testing Strategy

11.1 Backend Testing

Unit Tests

```
# applications/tests.py
from django.test import TestCase
from django.contrib.auth import get_user_model
from .models import Application

User = get_user_model()

class ApplicationModelTest(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            username='testuser',
            email='test@example.com',
            password='testpass123'
        )

    def test_create_application(self):
        app = Application.objects.create(
            user=self.user,
```

```
            company_name='Test Corp',
            position_title='Developer',
            status='Applied'
        )
        self.assertEqual(str(app), 'Test Corp - Developer')
```

API Tests

```
from rest_framework.test import APITestCase
from rest_framework import status

class ApplicationAPITest(APITestCase):
    def test_list_applications_unauthenticated(self):
        response = self.client.get('/api/applications/')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

11.2 Frontend Testing

Component Tests (Jest + React Testing Library)

```
import { render, screen } from "@testing-library/react";
import ApplicationCard from "./ApplicationCard";

test("renders application card", () => {
  const app = {
    company_name: "Google",
    position_title: "Engineer",
    status: "Interview",
  };

  render(<ApplicationCard application={app} />);
  expect(screen.getByText("Google")).toBeInTheDocument();
});
```

11.3 Testing Roadmap

Planned Tests:

- ☐ Unit tests for all models
- ☐ API endpoint tests
- ☐ Authentication flow tests
- ☐ Component unit tests
- ☐ Integration tests
- ☐ E2E tests with Playwright
- ☐ Performance testing
- ☐ Security testing

---

12. Future Enhancements

12.1 Short-term Goals (1-3 months)

Mobile Application Development Platform Options:

1. **React Native** (Recommended)
    - Reuse React knowledge and components
    - Share API integration logic
    - Single codebase for iOS + Android
    - Estimated timeline: 4-6 weeks
2. **Native Development**
    - iOS (Swift/SwiftUI)
    - Android (Kotlin/Jetpack Compose)
    - Best performance and native feel
    - Estimated timeline: 8-12 weeks

Mobile Features:

- Push notifications for application updates
- Offline mode with local storage
- Camera integration for document scanning
- Biometric authentication
- Share functionality

API Versioning

```
# Implement versioned API endpoints
/api/v1/applications/
/api/v2/applications/  # Future version
```

Benefits:

- Backward compatibility
- Gradual feature rollout
- Support multiple client versions

Enhanced Analytics

- Application success rate tracking
- Time-to-response metrics
- Company response patterns
- Interview conversion rates
- Visual charts and graphs

12.2 Medium-term Goals (3-6 months)

Advanced Features 1. Calendar Integration

- Sync interview dates with Google Calendar
- Automated reminders

- Deadline tracking

2. Email Integration

- Parse job application emails
- Auto-create applications from emails
- Track email responses

3. Resume Builder

- In-app resume editor
- Multiple resume versions
- Template library
- Export to PDF

4. Job Board Integration

- Import applications from LinkedIn
- Indeed API integration
- Glassdoor company reviews

5. Collaboration Features

- Share applications with mentors/coaches
- Feedback and comments system
- Group job search tracking

Performance Optimization Backend:

- Implement Redis caching
- Database query optimization
- API response pagination
- Celery for background tasks

Frontend:

- Code splitting and lazy loading
- Image optimization
- Service worker for offline support
- Progressive Web App (PWA) features

12.3 Long-term Goals (6-12 months)

AI/ML Integration 1. Resume Optimization

- AI-powered resume suggestions
- Keyword matching with job descriptions
- ATS compatibility checker

2. Job Matching

- ML-based job recommendations
- Skill gap analysis
- Salary prediction

3. Interview Preparation

- AI chatbot for practice interviews
- Common question database
- Company-specific prep materials

Enterprise Features 1. Team Plans

- University career centers
- Bootcamp student tracking
- Corporate outplacement services

2. Admin Dashboard

- User management
- Analytics and reporting
- Content moderation
- Subscription management

3. Premium Features

- Unlimited applications
- Advanced analytics
- Priority support
- Custom branding

12.4 Technical Debt & Improvements

Code Quality

- ☐ Increase test coverage to 80%+
- ☐ Implement comprehensive error handling
- ☐ Add request validation middleware
- ☐ Improve code documentation
- ☐ Set up pre-commit hooks

Security Enhancements

- ☐ Implement rate limiting
- ☐ Add CAPTCHA for registration
- ☐ Enable two-factor authentication
- ☐ Security audit and penetration testing
- ☐ GDPR compliance measures

DevOps Improvements

- ☐ CI/CD pipeline (GitHub Actions)
- ☐ Automated testing in pipeline
- ☐ Staging environment
- ☐ Monitoring and alerting (Sentry)
- ☐ Performance monitoring (New Relic)
- ☐ Automated backups

Accessibility

- ☐ WCAG 2.1 AA compliance
- ☐ Screen reader optimization
- ☐ Keyboard navigation
- ☐ High contrast mode
- ☐ Internationalization (i18n)

12.5 Monetization Strategy

Freemium Model Free Tier:

- Up to 20 active applications
- Basic dashboard
- Email support

Premium Tier ($9.99/month):

- Unlimited applications
- Advanced analytics
- Resume builder
- Priority support
- Calendar integration

Enterprise Tier (Custom pricing):

- Team collaboration
- Admin dashboard
- Custom branding
- Dedicated support
- SLA guarantees

---

13. Conclusion

13.1 Project Achievements

The Job Application Tracker successfully demonstrates:

Full-stack Development Proficiency

- Modern React frontend with Tailwind CSS v4
- Robust Django REST Framework backend
- Secure JWT authentication with OAuth integration

Software Engineering Best Practices

- RESTful API design
- Component-based architecture
- Database normalization
- Security-first approach

User-Centric Design

- Intuitive dashboard interface
- Responsive mobile design
- Demo mode for exploration
- Accessibility considerations

Production-Ready Features

- Environment-based configuration
- Docker containerization
- Static file optimization
- Error handling and validation

13.2 Technical Skills Demonstrated

Backend:

- Django ORM and migrations
- DRF serializers and viewsets
- JWT authentication
- Social OAuth integration
- File upload handling
- Database design and relationships

Frontend:

- React hooks and context
- React Router navigation
- Axios interceptors
- Tailwind CSS styling
- Component composition
- State management

DevOps:

- Docker containerization
- Environment configuration
- Static file serving
- CORS configuration
- Production deployment

13.3 Learning Outcomes

This project provided hands-on experience with:

1. **Authentication Complexity**: Implementing dual auth systems (email + OAuth)
2. **API Design**: Creating RESTful endpoints with proper security
3. **State Management**: Managing global auth state in React
4. **Database Relationships**: Designing normalized schemas
5. **Modern Tooling**: Vite, Tailwind v4, Django 5.2
6. **Problem Solving**: Debugging authentication issues and CORS problems

13.4 Next Steps

Immediate Actions:

1. Deploy to production environment
2. Implement comprehensive testing suite
3. Gather user feedback
4. Begin mobile app development

Future Vision: Transform this application into a comprehensive career management platform with AI-powered features, helping thousands of job seekers organize and optimize their job search journey.

Appendix

A. Project Links

- **GitHub Repository**: [Link to repository]
- **Live Demo**: [Link to deployed app]
- **API Documentation**: [Link to API docs]

B. References

- Django Documentation: https://docs.djangoproject.com/
- Django REST Framework: https://www.django-rest-framework.org/
- React Documentation: https://react.dev/
- Tailwind CSS: https://tailwindcss.com/
- Vite: https://vitejs.dev/

C. Acknowledgments

- **Technologies**: Django, React, Tailwind CSS, Vite
- **Icons**: Lucide React
- **Fonts**: Google Fonts (Inter)
- **Inspiration**: Modern job tracking applications

Report End

This comprehensive report documents the Job Application Tracker project as of February 8, 2026. The project continues to evolve with new features and improvements.