Store Locator Service - Final Project

Project Overview

Build a production-ready Store Locator API service for a multi-location retail business. The system supports public store search functionality and secure internal store management with role-based access control.

1. Store Search API (Public)

1.1 Search Endpoints

POST /api/stores/search

Support three search input types:

- Search by full address (e.g., "123 Main Street, Boston, MA")
- Search by postal code (e.g., "02101")
- Search by latitude & longitude (e.g., lat: 42.3601, lon: -71.0589)

Requirements:

- For address/postal code: integrate with free geocoding service (Nominatim or US Census)
- Convert address/postal to coordinates before searching
- Cache geocoding results to reduce API calls

1.2 Search Filters

Support the following query parameters:

- `radius_miles` (default: 10, max: 100)
- `services[]` (filter by store services - AND logic)
- `store_types[]` (filter by store type - OR logic)
- `open_now` (optional: filter stores currently open)

1.3 Response Format

Return results sorted by distance (nearest first) with:

- Store details (id, name, address, type, services, phone, hours, status)
- Distance in miles from search location
- Search metadata (location searched, applied filters)
- `is_open_now` boolean indicating current status

1.4 Distance Calculation Method

Required: Bounding Box Pre-filter + Haversine Formula

Implementation steps:

1. Calculate bounding box around search location
2. Use SQL WHERE clause to filter stores within box
3. Calculate exact distances for filtered stores
4. Sort by distance and apply radius filter

5. Return top results

Use `geopy` library for Haversine distance calculation.

Find calculation details in Appendix

1.5 Rate Limiting

Implement rate limiting:

- 100 requests per hour per IP address
- 10 requests per minute per IP address
- Return HTTP 429 with appropriate headers when exceeded

1.6 Caching

Implement caching for:

- Geocoding results (TTL: 30 days)
- Search results (TTL: 5-10 minutes, optional)

Use Redis or in-memory cache.

2. Store Management API (Internal/Authenticated)

2.1 CRUD Operations

POST /api/admin/stores - Create store

GET /api/admin/stores - List stores (with pagination)

GET /api/admin/stores/{store_id} - Get store details

PATCH /api/admin/stores/{store_id} - Partial update store

DELETE /api/admin/stores/{store_id} - Deactivate store (soft delete)

Partial Update Requirements:

- Support PATCH method for partial updates
- Only allow updates to specific fields:

- `name`

- `phone`

- `services`

- `status`

- `hours` (operating hours)

- Fields NOT allowed to update:

- `store_id` (immutable)

- `latitude` / `longitude` (use dedicated endpoint if needed)

- `address` fields (use dedicated endpoint if needed)

- Validate only the fields provided in request
- Return updated store data

Other Requirements:

- All endpoints require authentication
- DELETE sets `status = "inactive"` (do not physically delete)
- Validate all inputs
- Auto-geocode address if coordinates not provided (on create)

2.2 Authentication & Authorization

JWT Two-Token Pattern:

- Access token: 15 minute expiry
- Refresh token: 7 day expiry
- Store refresh tokens in database for revocation

Required Endpoints:

- `POST /api/auth/login` - Returns access + refresh tokens
- `POST /api/auth/refresh` - Returns new access token
- `POST /api/auth/logout` - Revokes refresh token

Token Payload:

```
{ "user_id": "U001", "email": "user@company.com", "role": "admin", "exp": 1735123456 }
```

Role-Based Access Control (RBAC):

Three roles:

Admin:

- Full access to all endpoints
- Can manage stores and users
- Can perform batch imports

Marketer:

- Can manage stores (create, update, deactivate)
- Can perform batch imports
- Cannot manage users

Viewer:

- Read-only access to stores
- Cannot modify anything

Implementation:

- Create users, roles, permissions tables
- Implement decorator/middleware for permission checking
- Hash passwords with bcrypt

2.3 Batch CSV Import

POST /api/admin/stores/import

Accept CSV file upload with store data.

Import Behavior: Create or Update (Upsert)

- If `store_id` exists in database: **UPDATE** the existing store
- If `store_id` does NOT exist: **CREATE** new store
- This allows CSV to be used for both initial load and ongoing updates

Requirements:

- Validate CSV structure and headers
- Validate each row (required fields, data formats, coordinate ranges)
- Auto-geocode if coordinates missing
- Use database transaction (all-or-nothing)
- Return detailed report:

- Total rows processed

- Successfully created (new stores)

- Successfully updated (existing stores)

- Failed (with error messages and row numbers)

CSV Processing:

- Choose between Python built-in `csv` module or `pandas`

2.4 User Management (Admin Only)

POST /api/admin/users - Create user

GET /api/admin/users - List users

PUT /api/admin/users/{user_id} - Update user (role, status)

DELETE /api/admin/users/{user_id} - Deactivate user

3. Database Requirements

3.1 Core Tables

Design tables for:

- stores
- store_services (many-to-many)
- users
- roles
- permissions
- role_permissions (junction table)
- refresh_tokens

Important: Infer the stores table structure from the CSV file format below.

3.2 Required Indexes

Create indexes on:

- `stores(latitude, longitude)` - composite index for geographic search
- `stores(status)` - partial index for active stores
- `stores(store_type)` - for filtering
- `stores(address_postal_code)` - for ZIP search
- `users(email)` - for login
- `refresh_tokens(token_hash)` - for token lookup

4. Sample Dataset

4.1 Stores CSV File Structure

A CSV file with 1,000 stores for batch import will be provided.

A CSV file with 50 stores for initial seeding will be provided.

File: stores_1000.csv

File: stores_50.csv

Required Columns (EXACT order and names):

```
store_id,name,store_type,status,latitude,longitude,address_street,address_city,address_state,address_postal_code,address_country,phone,services,hours_mon,hours_tue,hours_wed,hours_thu,hours_fri,hours_sat,hours_sun
```

Column Specifications:

Column	Format	Example	Notes
store_id	S{0001-1000}	S0001, S0042	Unique identifier
name	String	Boston Downtown Store	Store name
store_type	Enum	flagship, regular, outlet, express	Must be one of these values
status	Enum	active, inactive, temporarily_closed	Must be one of these values
latitude	Decimal	42.3555	Range: -90 to 90
longitude	Decimal	-71.0602	Range: -180 to 180
address_street	String	100 Cambridge St	Street address
address_city	String	Boston	City name
address_state	String(2)	MA	Two-letter state code
address_postal_code	String(5)	02114	5-digit ZIP code
address_country	String(3)	USA	Country code
phone	String	617-555-0100	Format: XXX-XXX-XXXX
services	String	pharmacy|pickup|optical	Pipe-separated (no spaces)
hours_mon	String	08:00-22:00	Format: HH:MM-HH:MM or "closed"
hours_tue	String	08:00-22:00	Format: HH:MM-HH:MM or "closed"
hours_wed	String	08:00-22:00	Format: HH:MM-HH:MM or "closed"
hours_thu	String	08:00-22:00	Format: HH:MM-HH:MM or "closed"
hours_fri	String	08:00-22:00	Format: HH:MM-HH:MM or "closed"
hours_sat	String	09:00-21:00	Format: HH:MM-HH:MM or "closed"
hours_sun	String	10:00-20:00	Format: HH:MM-HH:MM or "closed"

Services Format:

- Pipe-separated values (e.g., `pharmacy|pickup|optical`)

- No spaces between values
- Available services: pharmacy, pickup, returns, optical, photo_printing, gift_wrapping, automotive, garden_center

Hours Format Rules:

- Open hours: `HH:MM-HH:MM` (24-hour format, e.g., "08:00-22:00")
- Closed: `closed` (lowercase)
- Must validate time format and logical ordering (open_time < close_time)

Example Row:

```
S0001,Boston Downtown Store,flagship,active,42.3555,-71.0602,100 Cambridge St,Boston,MA,02114,USA,617-555-0100,pharmacy|pickup|optical,08:00-22:00,08:00-22:00,08:00-22:00,08:00-22:00,08:00-23:00,09:00-23:00,10:00-20:00
```

Dataset Characteristics:

- 1,000 stores across all 50 US states
- Store type distribution: flagship (5%), regular (70%), outlet (20%), express (5%)
- Status distribution: active (95%), temporarily_closed (3%), inactive (2%)
- Geographic distribution by population density
- Valid city/state/ZIP combinations

4.2 User Seed Data

Create at least 3 test users:

- 1 Admin (full access)
- 1 Marketer (store management only)
- 1 Viewer (read-only)

Default password: `TestPassword123!` (must be changed on first login)

5. Technical Requirements

5.1 Framework Choice

Choose ONE of the following Python web frameworks:

- Flask
- FastAPI
- Django + Django REST Framework

5.2 Required Dependencies

Core:

- Python 3.10+
- PostgreSQL (plain, no PostGIS)
- Your chosen framework
- SQLAlchemy or Django ORM
- geopy (for distance calculation)
- PyJWT (for JWT tokens)
- bcrypt (for password hashing)

CSV Processing (choose one):

- Python built-in `csv` module
- pandas

Caching:

- Redis (recommended for deployment)
- In-memory cache (acceptable for development)

Optional:

- Alembic (for migrations if using Flask/FastAPI)
- pytest (for testing)
- python-dotenv (environment variables)

5.3 API Standards

- RESTful design principles
- Proper HTTP status codes (200, 201, 400, 401, 403, 404, 429, 500)
- JSON request/response format
- OpenAPI/Swagger documentation
- Consistent error response format

5.4 Security Requirements

- Password hashing (bcrypt)
- JWT token authentication
- Token expiration and refresh
- Rate limiting on public endpoints
- Input validation on all endpoints
- SQL injection prevention (use parameterized queries)
- CORS configuration

6. Testing Requirements

6.1 Required Tests

Implement basic tests covering:

Unit Tests:

- Distance calculation (Haversine formula)
- Bounding box calculation
- Hours parsing and validation
- Password hashing/verification

API Tests:

- Search by address/ZIP/coordinates
- Filter by radius, services, store types
- Authentication (login, refresh, logout)
- Authorization (role-based access)
- CRUD operations
- CSV import validation

Integration Tests:

- End-to-end search flow
- Authentication + protected endpoint access
- CSV import with geocoding

Test Data:

- Use subset of store data (10-20 stores)
- Create test users for each role
- Mock external API calls (geocoding)

6.2 Testing Framework

Choose appropriate framework for your web framework:

- Flask: pytest or unittest
- FastAPI: pytest with TestClient
- Django: Django TestCase

6.3 Coverage Target

- Minimum 60% code coverage
- Focus on critical business logic

7. Deployment Requirements

7.1 Deployment Platforms

Choose a platform and deploy all components:

Platform Options:

- Railway
- Render
- Heroku
- AWS/GCP/Azure
- DigitalOcean

7.2 Required Components

Deploy the following:

1. Application Server

- Your Python web application
- Environment variables configured
- Production-ready web server (Gunicorn for Flask/FastAPI, uWSGI for Django)

2. PostgreSQL Database

- Production database instance
- Schema created with migrations
- Sample data loaded (1000 stores)
- Seed users created

3. Redis Cache (if using)

- Redis instance deployed
- Connected to application
- Verify caching is working

7.3 Deployment Checklist

- [ ] Application runs and serves requests
- [ ] Database is accessible and populated
- [ ] Redis/cache is working (if implemented)
- [ ] Environment variables properly configured
- [ ] HTTPS enabled
- [ ] CORS configured correctly
- [ ] Rate limiting functional
- [ ] API documentation accessible
- [ ] Health check endpoint working
- [ ] Logs accessible for debugging

Example Test Credentials Format:

```
Admin User: Email: admin@test.com Password: AdminTest123! Marketer User: Email: marketer@test.com Password: MarketerTest123!
```

8. Deliverables

8.1 Code Repository

- Clean, organized project structure
- requirements.txt with all dependencies
- .env.example for configuration
- Database migration scripts
- README.md (see below)

8.2 Documentation

README.md must include:

- Project description
- Framework choice
- CSV processing choice (built-in vs pandas)
- Setup instructions (database, dependencies, environment)
- How to run locally
- How to run tests
- API endpoint documentation (or link to Swagger)
- Authentication flow explanation
- Distance calculation method explanation
- Deployment information (URL, credentials, platform)

Additional documentation:

- Database schema
- Architecture overview
- Sample API requests/responses

8.3 Testing

- Test suite implemented
- All tests passing
- Coverage report generated
- Instructions to run tests in README

8.4 Deployment

- Application deployed and accessible
- Database deployed with sample data
- Cache deployed (if using Redis)
- All components working together
- Health check endpoint responding

9. Bonus Features (Optional)

9.1 Project Management with Kanban Board

Break down the project into tickets and track progress using a free Kanban board tool:

Recommended Tools:

- Trello
- GitHub Projects
- Notion
- Jira (free tier)

Suggested Ticket Categories:

Setup & Infrastructure:

- Set up development environment
- Initialize database schema
- Configure authentication system
- Set up testing framework
- Configure deployment environment

Store Search Features:

- Implement search by coordinates
- Implement search by address/ZIP with geocoding
- Add filtering (radius, services, store types)
- Implement caching for geocoding
- Add rate limiting

Store Management:

- Create store CRUD endpoints
- Implement partial update (PATCH)
- Add CSV import endpoint
- Validate CSV data
- Handle create/update logic

Authentication & Authorization:

- Implement JWT login/refresh/logout
- Create RBAC system
- Add permission middleware
- User management endpoints

Testing & Documentation:

- Write unit tests
- Write API tests
- Write integration tests
- Create API documentation
- Write README

Deployment:

- Deploy database
- Deploy application
- Deploy cache

- Configure environment
- Verify production

Deliverable:

- Include link to your Kanban board in README
- Export final board state as screenshot or PDF
- Document your project management approach

9.2 Store Ratings & Reviews

Add review system:

- Users can submit ratings (1-5 stars) and reviews
- Calculate average rating per store
- Filter search by minimum rating
- Review moderation (flag/remove inappropriate content)

9.3 Advanced Features

- Driving distance calculation (Google Maps API integration)
- Advanced hours handling (holiday hours, special events)
- Multi-language support
- Analytics dashboard
- Export search results to CSV
- Email notifications for critical events
- Docker containerization
- CI/CD pipeline

11. Getting Started

1. Choose your web framework (Flask/FastAPI/Django)
2. Set up local development environment
3. Set up PostgreSQL database locally
4. Create database schema and tables
5. Load provided CSV file (1000 stores)
6. Create seed users with different roles
7. Implement search API first (test with public data)
8. Add authentication and admin features
9. Implement partial update (PATCH) endpoint
10. Implement CSV import with create/update logic
11. Add rate limiting and caching
12. Write tests
13. Deploy to chosen platform
14. Document everything in README

Appendix: Distance Calculation Methods

Method 1: Simple Approach (Not Recommended)

Fetch all stores, calculate distances in Python, filter and sort.

Method 2: Bounding Box + Haversine (Required)

Step 1: Calculate Bounding Box

```
latitude_delta = radius_miles / 69.0 longitude_delta = radius_miles / (69.0 * cos(latitude_radians)) min_lat = search_lat - latitude_delta max_lat = search_lat + latitude_delta min_lon = search_lon - longitude_delta max_lon = search_lon + longitude_delta
```

Step 2: SQL Filter

```
WHERE latitude BETWEEN min_lat AND max_lat AND longitude BETWEEN min_lon AND max_lon AND status = 'active'
```

Step 3: Calculate Exact Distance (Python)

```
from geopy.distance import geodesic for store in filtered_stores: distance = geodesic( (search_lat, search_lon), (store.latitude, store.longitude) ).miles
```

Step 4: Filter and Sort

```
results = [s for s in stores if s.distance <= radius] results.sort(key=lambda x: x.distance)
```

Method 3: SQL Haversine Formula (Bonus)

Haversine Formula in SQL:

```
SELECT *, (3959 * acos( cos(radians(?)) * cos(radians(latitude)) * cos(radians(longitude) - radians(?)) + sin(radians(?)) * sin(radians(latitude)) )) AS distance FROM stores WHERE latitude BETWEEN ? AND ? AND longitude BETWEEN ? AND ? AND status = 'active' HAVING distance <= ? ORDER BY distance LIMIT 20;
```

Parameters: search_lat, search_lon (3 times), min_lat, max_lat, min_lon, max_lon, radius

Constant: 3959 = Earth's radius in miles (use 6371 for kilometers)