CenterPlate Restaurant Location Algorithm Documentation

If you want to tinker with the restaurant scoring system, open Google Colab and throw in centerplate_restaurant_finding_demo.ipynb, which should be in docs folder.

Algorithm Flow

1. Input Validation & Processing

Files: midpointService.js

- validateCoordinates(): Ensures coordinates are valid arrays of [lat, lng] pairs within proper bounds (-90 to 90 for latitude, -180 to 180 for longitude)
- validateFilters(): Validates restaurant filter arrays (e.g., [['Italian'], ['$$']])
- validateOptions(): Checks optional parameters like minimum acceptable scores and restaurant counts

2. Midpoint Calculation Methods

Files: calcGeoMidpoint.js, calcSmartMidpoint.js

The algorithm employs two distinct approaches to find optimal meeting points:

Geometric Midpoint (calcGeoMidpoint.js)

- **Function**: geometricMidpoint(coordinates)
- **Method**: Simple mathematical average of all participant coordinates
- **Metrics**: Calculates average distance, maximum distance, and distance variance from the midpoint
- **Use Case**: Simple calculation for evenly distributed participants. Sometimes, somehow, the best restaurants will be in the mathematically perfect midpoint.

// Example: For coordinates [[40.7128, -74.0060], [40.7589, -73.9851]]
// Returns: midpoint at [40.73585, -73.99555] with distance metrics

Smart Midpoint (calcSmartMidpoint.js)

- **Function**: smartMidpoint(coordinates)
- **Method**: Population-weighted algorithm that considers major urban centers
- **Data Source**: Fetches US cities with 1000+ population from OpenDataSoft API
- **Algorithm**:
    1. **Grid Search**: Instead of just calculating one center point, it tests **25 different locations** in a grid pattern:
        - Takes the basic geometric center
        - Tests points in a 5x5 grid around it (from 1 degree north/south/east/west)
        - Each grid square is 0.5 degrees apart (roughly 35 miles)

2. **Scoring System**: For each of those 25 test points, it calculates two scores:
    - **Hub Score (40% weight)**: How close is this point to major cities? Closer to big cities = higher score
    - **Density Score (30% weight)**: How close is this point to all the participants? More central = higher score
    - **Total Score**: `0.4 × hub_score + 0.3 × density_score`
3. **Safety Check**: After finding the best scoring point:
    - Checks if any participant would have to travel more than 500km (310 miles)
    - If yes, it throws out that result and picks the nearest major city instead
    - This prevents recommending a spot in the middle of nowhere when people are really far apart e.g. NYC and LA

- **Use Case**: Better results for widely distributed participants or when proximity to urban centers matters

3. City Data Integration

Files: midpointUtils.js

fetchAllCities() Function

- **API**: OpenDataSoft Geonames dataset (geonames-all-cities-with-a-population-1000)
- **Scope**: All US cities with population ≥ 1000
- **Data**: City name, coordinates, and population for each location
- **Usage**: Powers both smart midpoint calculation and fallback city selection

4. Restaurant Scoring Algorithm

Files: midpointUtils.js, getRestaurants.js

Primary Scoring: scoreLocationRestaurants()

This is the **core algorithm** that determines location quality:

Input Processing:

- Takes coordinate pairs and filter arrays

- Generates filter combinations (progressive: first filter alone, then first+second, etc.)
- Uses Foursquare Places API v3 for restaurant data

Scoring Formula:

```
// With filters:
score = (0.6 * distance_factor + 0.4 * filter_match) * 100

// If nobody inputs any preferences:
score = (distance_factor) * 100

// Where distance_factor = 1 / (1 + distance_km / 5)
```

Key Features:

- **Deduplication**: Uses Set to prevent duplicate restaurants across multiple searches
- **Comprehensive Data**: Fetches restaurant names, addresses, photos, categories, and coordinates
- **Distance Calculation**: Uses geolib for accurate geographic distance calculation
- **Top Results**: Returns top 10 highest-scoring restaurants per location

Location Comparison: `findBestRestaurantLocation()`

Files: `getRestaurants.js`

Process:

1. **Primary Search**: Tests both geometric and smart midpoints
2. **Fallback Strategy**: If no location meets minimum criteria:
    - Sorts all US cities by population (uses top 300)
    - Calculates distance from midpoint center
    - Tests 5 nearest major cities
    - Selects best-performing city as "Nearest Major City" result

Scoring Criteria:

- Minimum acceptable score (default: 50/100)
- Minimum restaurant count (default: 5)
- Prioritizes locations with both high scores AND sufficient restaurant options

5. Image Integration

Files: `restaurantImages.js`

`getRestaurantImages()` Function

- **API**: Foursquare Places API v3 photos endpoint
- **Features**:
  - Multiple image sizes (original, thumbnails, various preset sizes)
  - Metadata including creation date, dimensions, visibility
  - Error handling for restaurants without photos
- **Integration**: Called separately via /place-image/:id endpoint

API Integration Details

Foursquare Places API v3

Configuration: API key stored in environment variables (FOURSQUARE_API_KEY)

Endpoints Used:

1. **Search**: /v3/places/search - Primary restaurant discovery
2. **Photos**: /v3/places/{place_id}/photos - Restaurant image retrieval

Search Parameters:

- ll: Latitude,longitude coordinates
- categories: "13065" (Restaurant category code)
- limit: 50 restaurants per query
- fields: Comprehensive data including photos, location, categories
- query: Dynamic based on user filters

OpenDataSoft Geonames API

Endpoint: https://public.opendatasoft.com/api/records/1.0/search/ **Dataset**: geonames-all-cities-with-a-population-1000 **Filter**: refine.country_code=US (US cities only)

API Endpoints

Primary Endpoint: POST /api/midpoint/find

File: midpointRoutes.js **Controller**: findOptimalLocationController

Request Body:

```
{
  "coordinates": [[40.7128, -74.0060], [34.0522, -118.2437]],
  "filters": [["Italian"], ["$$"]],
  "options": {
    "minAcceptableScore": 50,
    "minRestaurantCount": 5
  }
}
```

Response:

```
{
  "bestScore": {
    "totalScore": 85.2,
    "restaurantCount": 23,
    "avgDistance": 2.1,
    "bestRestaurants": [/* top 10 restaurants */],
    "location": [40.7358, -73.9955]
  },
  "bestMethod": "smart"
}
```

Image Endpoint: GET /api/midpoint/place-image/:id

File: midpointRoutes.js **Controller**: getPlaceImagesController **Returns**: Array of image objects with URLs and metadata

Performance Considerations

API Efficiency

- **Smart Caching**: Deduplicates restaurants across multiple filter searches
- **Progressive Filtering**: Builds filter combinations incrementally rather than testing all permutations
- **Limited Scope**: Tests maximum 7 locations (2 midpoints + 5 fallback cities)

Error Handling

Files: All service files include comprehensive error handling

- Network failures gracefully handled with informative error messages
- Input validation prevents malformed requests
- Logging throughout for debugging and monitoring

Rate Limiting Awareness

- **Foursquare Free Tier**: 100,000 calls/month
- **Typical Usage**: ~10-15 API calls per location request
- **Estimated Capacity**: ~6,000-10,000 location requests per month on free tier during debugging and small beta testing periods

Address Utilities

File: addressUtils.js

Additional Support Functions:

- **addressToCoordinates()**: Converts street addresses to coordinates using Nominatim (OpenStreetMap)
- **coordinatesToAddress()**: Reverse geocoding for coordinate-to-address conversion
- **Integration**: Supports user-friendly address input instead of raw coordinates

Authentication & Security

File: midpointRoutes.js

- **Firebase Authentication**: All endpoints protected with authenticateFirebaseToken middleware
- **Input Validation**: Custom validators prevent malicious input
- **Environment Variables**: API keys secured in environment configuration

Shoot me an email at connortierneywork (at) gmail.com if you have any questions about this.