Restaurants Rest API Roadmap

Contents

[Introduction 1](#_Toc111495185)

[Phase 1 – Refactoring 1](#_Toc111495186)

[Database Normalization 1](#_Toc111495187)

[No-Sql Database 1](#_Toc111495188)

[Encrypting Configuration Values 2](#_Toc111495189)

[Allow server settings to be configured 2](#_Toc111495190)

[More logging options 2](#_Toc111495191)

[Update Jooq queries 2](#_Toc111495192)

[Phased Two – Additional endpoints 2](#_Toc111495193)

[Allow users to search based on their location 2](#_Toc111495194)

[Allow users to search based on cuisine 2](#_Toc111495195)

[Allow users to search based on last inspection date 2](#_Toc111495196)

[Combined options 2](#_Toc111495197)

[Phase 3 – Admin tools 3](#_Toc111495198)

[Phase 4 – User tools 3](#_Toc111495199)

[Phase 5 – Different Companies 3](#_Toc111495200)

# Introduction

This document outlines the recommended path to updating the restaurants Rest API, so it can reach its full potential. The document splits the updates into phases and each phase should be completed in order, to ensure work doesn’t need to be repeated and tasks can be completed in a timely manner.

# Phase 1 – Refactoring

The first phase is refactoring the database and some of the missing configuration options. Fixing these first will reduce the refactoring work later and make development easier going forward.

### Database Normalization

Currently the data is stored in one large table, which will struggle under-load as more records are added. The best thing to do is normalize the tables, so the data is spread out more and indexes can be used to speed up performance. While the primary key fields will automatically have indexes, other indexes may be required, and these should be created based on usage. There’s no point indexing the company name if users are searching by grade and location.

#### Proposed tables

**Company table:** This table contains the common company data. Name, head office/main premises.

**Restaurant Location Table:** This table contains the address, phone number and latitude & longitude values. Linking to the company table via a company\_id key, we can store multiple locations in a sensible wa**y**

**Inspection details table:** This table contains information regarding the inspection details. This is linked to the ***Restaurant Location Table*** by a location\_id field. This allows us to store historical inspection data and return the results of a specified timeframe.

**Cuisine table:** This is a simple lookup table. The different types of cuisine can be linked to the company table, so a cuisine can be linked to the relevant companies the serve that cuisine and then link to those locations. Careful thought will need to be given to any company that may have specialized locations, so they don’t get included in the wrong category. For example, KFC known for serving chicken-based meals, but if they had a few restaurants that only served Vegan dishes, these would need to be filtered from the list. So multiple company records may be needed to split specialized locations. If it was deemed unfeasible to split this way, linking cuisine to the location table would also be an option.

### No-Sql Database

Converting to a no-sql database may also be an option worth considering. The json data could follow the new tables mention in the previous section. It would just be json objects rather than tables. The company details would form the parent object, with the location and cuisine object being children.

### Encrypting Configuration Values

To ensure security best practice, sensitive values like usernames, passwords and possibly server details, should be encrypted in any config files. This ensures anyone looking at config files cannot access sensitive information they shouldn’t. Jasypt can be added to the maven dependencies so values can be stored in encrypted form

### Allow server settings to be configured

Currently spring boot will try and run the Rest API on port 8080 and if it can’t, will try another port. While this makes development simple, configuring docker containers, or similar things can be made easier if the host, port, and similar setting can be configured at compile time. Creating a config file with these settings will ensure future improvements can be made with the minimum of effort.

### Allow database connection pooling

Currently Spring manages the database connection. For better performance Tomcat can create a connection pool and the Rest API can the use that. This will allow finer control over number of connections and also allow other java apps to share the same pool if load allows.

### More logging options

Currently all logs are written to the console and a log file. To allow for future growth it’s probably worth creating a database logger that writes all logs to the database. This allows error to be searched for easily and also allow database triggers to fire off scripts when certain log entries are created. It may also be worth creating an email logger that emails a specified list whenever certain error are encountered. This allows developers/management to know about errors, before customers start complaining (about that particular error anyway).

### Update Jooq queries

Once the database has been updated, the existing Jooq queries will need to be updated. As there are only two queries, this shouldn’t take long. A new Model class will need to be created, so we can store the data from multiple tables in a single class.

# Phased Two – Additional endpoints

Phase two concentrates on expanding the functionality of the existing Rest API, allowing users to find an appropriate restaurant in more ways. This is phased two, as doing before phase 1 will simply mean more code refactoring will need to be done after the database is updated. The list below can be used as a starting point, but the best people to decide this is probably user requests.

### Allow users to search based on their location

A new endpoint should be created that accepts the users current, latitude and longitude, along with a radius in miles or kilometres. The code then searches the database for all locations within that range and returns the list.

### Allow users to search based on cuisine

An endpoint should be created that allows the users to specify a single cuisine, or multiple choices and then have all restaurant locations that match returned.

### Allow users to search based on last inspection date

Users may want to see Restaurants that have been inspected recently. Having an A rating 4 years ago, doesn’t mean everything is still healthy. The endpoint can accept a date and return everything inspected on or after that date.

### Combined options

Most of the above criteria can be combined to filter the results even more, so where appropriate create endpoints that cater for these options. An example would be location and grade.

# Phase 3 – Admin tools

A frontend to allow admins to add, edit and delete Restaurant data can be created. As restaurants go out of business, or get bought by other companies, the data in the database will go out of date. Having a way to update or remove data will become essential. Also inspection data will need adding.

# Phase 4 – User tools

Having the ability to have users store their favourite restaurants and add reviews, will help make our data more useable and encourage user interaction. Perhaps the user can rate the restaurants themselves and other users can opt-in to see their ratings. Making ratings opt-in will help address review bombing and companies buying reviews.

# Phase 5 – Different Companies

Now we have conquered the restaurant world, we can consider adding other types of companies to the Rest API. The company table can have a company\_type\_id added that defines the type of company and alongside the cuisine table, we could have a product table. This table will list products and will link to the company table in a many to one relationship. Further tables can be added as appropriate, so that users can filter to a high level. The product table may contain an entry for clothing, but another table can be created that splits clothing down further e.g., Shoes, sportwear etc.