Task 5: Please don't sue me apple.

Some hotshot media mogul has heard of your newly acquired skills in Java Spring. They have contracted you to stride on the edge of copyright glory and start re-making iTunes, but under a different name. They have spoken to lawyers and are certain a working prototype should not cause any problems and ensured that you will be safe. The lawyer they use is the same that Epic has been using, so they are familiar with Apple.

For the time being the working prototype is a simple UI built in Thymeleaf with the rest of the application being exposed as a regular REST API to be possibly consumed by a mobile app in the future.

They have provided a database which emulates the iTunes functionality, it is a SQLite database for simplicity (SQLite JDBC driver). It is called chinook and can be found here. There are some ERDs of the database drawn here and here. They serve to guide you when writing queries.

Requirements

Make use of Spring Boot and Spring Initializr to create a web application. This web application must contain Thymeleaf as a templating engine.

You are to create several views with Thymeleaf.

1. The home page view, showing the 5 random artists, 5 random songs, and 5 random genres. This home page contains a search bar which is used to search for tracks. The search bar should not be empty, meaning you can't have an empty search criterion.
2. The search results page will show the query the user has made, i.e. Search results for "Never gonna give you up". Underneath this, the results will be shown for the search. The search results should show a row where the track name, artist, album, release year, and genre are shown. The search should also be case insensitive.

In addition to these two simple views, there must be a portion of your application which is dedicated to expose REST endpoints. These endpoints will be used by other applications, so returning the data in JSON is required. These endpoints must be on a /api/ sub directory in your

applications structure. Meaning, “/” and “/search?term=foo” are for the Thymeleaf pages and “/api/bar” is for the REST endpoints.

Your API endpoints should cater to the following functionality:

1. Read all the customers in the database, this should display their: Id, first name, last name, country, postal code, and phone number.
2. Add a new customer to the database, with a random employee assigned support rep (you can hard code this to be any of the employees).
3. Update an existing customer.
4. Return the number of customers in each country, ordered descending (most number of customers to least). i.e. USA: 10000,
5. Customers who are the highest spenders (total in invoice table is the largest), ordered descending.
6. For a given customer, their most popular genre (in the case of a tie, display both). Most popular in this context means the genre that corresponds to the most tracks from invoices associated to that customer.

The endpoints should be designed with best practices in mind. The endpoints should be named appropriately, remember, nouns not verbs.

Provide a collection of API calls made in Postman to test the endpoints (done by creating a collection and exporting it as JSON).

Finally, the application must be published on Heroku. Bonus points if you can publish a docker container of your app on Heroku.

Submission

Git repo with all the source files. Provide a link to the Heroku application in the readme. The exported Postman collection can also be included anywhere in the project – but it must make sense why it is there. Please keep in mind that best practices should be followed for every aspect of the application.

Learning outcomes

Configuring REST endpoints with Spring.

Using Thymeleaf to create template html pages with templated data.

Publishing Spring applications to Heroku.

Demonstrate proper endpoint naming conventions.

SQL queries with JOINs, ORDERBY, GROUPBY, and MAX.

Hints

Use @Controller for the controllers that are used with Thymeleaf and @RestController for the API endpoints – this saves a lot of configuration.

Create POJOs for each query data structure you expect, be wary of over-posting (posting unneeded fields).

Don't be afraid to go deeper with endpoint naming hierarchies, its perfectly fine to have an endpoint like: /api/customers/:customerId/popular/genre. This is actually the preferred naming convention; this way clients are less likely to need documentation to explain what is going on. The reason /popular/ is included in that URI is because a customer can also have a most popular artist, which is just a simple change from genre to artist and the endpoint is still perfectly understandable by the client with no extra assistance.

For handling database interaction, investigate the Repository pattern or how to make Data access classes. Essentially, you want a class (customerRepository for example) that has methods (getCustomers for example) that moves to SQL execution away from your controllers. This is the intention with the MVC pattern and promotes good programming practices.

Look into how to return specific status codes with the response data.