Messaging with Kafka

Overview

In this lab you'll enhance your "online retailer" application so that it sends and receives Kafka messages when important business events occur.

IntelliJ starter project

If you're happy to continue where you left off in the previous lab, use the following project:

- **`student\student-online-retailer`**

If you'd prefer a fresh start, use the solution project from the previous lab instead:

- **`solutions\testing`**

IntelliJ solution project

The solution project for this lab is located here:

- **`solutions\solution-kafka`**

Instructions

Implement Kafka message queuing in the application, as follows:

- If you haven't already done so, install Kafka on your computer and tweak the Zookeeper and Kafka configuration as described in the chapter. Then start Zookeeper and Kafka.
- In your Spring Boot application, add the Kafka dependency in your pom file.
- Enhance the **`ProductSuggestionController`** class so that it publishes messages to a topic named **`"product_suggestions_topic"`** whenever anything interesting happens. For example:
    - If the user inserts a new product suggestion, publish this message:
      Key **`"inserted"`**
      Value product suggestion details
    - If the user modifies a product price value, publish this message:
      Key **`"modifiedPrice"`**
      Value ID and new price
    - If the user modifies a product sales value, publish this message:
      Key **`"modifiedSales"`**
      Value ID and new sales
    - If the user increases the price for popular products, publish this message:
      Key **`"increasedPriceForPopularProducts"`**
      Value sales threshold
    - If the user deletes all product suggestions, publish this message:
      Key **`"deletedAll"`**
      Value (empty value)
- Define a component class named **`ProductSuggestionTopicListener`** that listens for all messages published to the **`"product_suggestions_topic"`**. Whenever it receives a message, simply display the message on the screen to indicate it has been received. In a real application you would do something semantically important at this juncture.

Build and run the application in IntelliJ, then use Swagger to ping various REST endpoints to insert, modify, and delete product suggestions. Then take a look in the IntelliJ console and verify messages have been published and received on the Kafka topic, such that the IntelliJ console displays the messages received.

When you're all done, take a look at the *solution* project in IntelliJ. We've implemented 2 versions of the "product suggestions" REST controller - one that supports Kafka messaging, and one that doesn't. We've annotated the two controller classes with **`@Profile`**, so that the application either uses (or doesn't use) Kafka depending on which active profile is set. Now take a look in **`application.properties`** to see what profile is active. See what happens if you change this property. This is a good example of the use of profiles to influence how Spring Boot applications work in practice.