The Prices Evaluator API

Domain Definition

In the e-commerce database of a retail company we have the **PRICES** table that reflects the final price ("PVP") and the rate that applies to a product of a brand between certain dates. The relevant fields are explained below:

- BRAND_ID: foreign key of the group string (1 = ZARA).
- START_DATE, END_DATE: range of dates for which the price is valid.
- PRICE_LIST: Identification of the applicable price list.
- PRODUCT_ID: Product code identifier.
- PRIORITY: Price application disambiguator. If two prices coincide in a date range, the one with the higher priority (higher numerical value) is applied.
- PRICE: Final price to sell.
- CURR: Currency ISO.

PRICES Table example sample:

BRAND_ID	START_DATE	END_DATE	PRICE_LIST	PRODUCT_ID	PRIORITY	PRICE	CURR
1	2020-06-14-00.00.00	2020-12-31-23.59.59	1	35455	0	35.50	EUR
1	2020-06-14-15.00.00	2020-06-14-18.30.00	2	35455	1	25.45	EUR
1	2020-06-15-00.00.00	2020-06-15-11.00.00	3	35455	1	30.50	EUR
1	2020-06-15-16.00.00	2020-12-31-23.59.59	4	35455	1	38.95	EUR

Solution requested

It is required to build an application/service in SpringBoot that provides a REST query endpoint such that:

INPUT

Accepts as input parameters: application date, product identifier, string identifier.

OUTPUT

Returns as output data: product identifier, string identifier, rate to apply, application dates and final price to apply.

EXTRA INFO:

You must use an in-memory database (type h2) and initialise it with the data of the example (it is possible to change the names of the fields and add new ones if necessary, the type of data chosen for each field is left to the free choice of the solution designer/developer).

Initial design/development commitment

The following points will be attempted in the construction of the proposed service:

- Prioritise working code over complexity (incremental growth).
- Maintain a clear and clean code structure. Intended for use:
    - API First approach.
    - Hexagonal architecture.
    - DDD.
    - SOLID principles.
    - Software Design Patterns.
- Avoid:
    - Spaghetti code.
    - Dead code.
    - Unnecessary or unused functionality.
    - Extra dependencies that add no value.
- Gradle will be used for dependency management (if Maven is preferred, this could easily be changed, if there is time we will try to create a working branch with Maven).
- Docker will be used.
- Unit tests will be implemented using JUnit (and Mockito), and we will try to test the cases of the declaration both as integration tests and as tests using Postman (providing the Postman JSON collection).
- A README.md file will be written with full project information and instructions on how to install/run the project.
- Exception handling will be provided.