05/04/2025

Laravel Senior Developer Test

Mohamad Nashar

# Database Design:

I made a slight change to the suggested schema, splitting the price\_lists table into two tables as follows:

* products (id, name, base\_price, description).
* price\_list\_items (id, product\_id, price\_list\_id, price)
* price\_lists (id, country\_id, currency\_id, start\_date, end\_date, priority)
* countries (id, code, name)
* currencies (id, code, name)

This improves data normalization, scalability and flexibility and prevents data repetition if a price list is applied on a lot of products.

# Models:

The first step is to create all the models we need, in our case we need a model for each table above, along with the models we add the migrations.

In each of the models we write the relation methods between the models along with the fillable property, which is used to prevent mass assignment and give permission to only edit specific fields of the model.

# Factories and Seeders:

These are used to manipulate the database and add some data to it.

The country seeder reads all the countries and currencies from countries\_api.json file. I created this file from the following API:

<https://restcountries.com/v3.1/independent?status=true>

The API is not too stable and fails a lot, therefore I preferred to save all the response into a json file and read the data from it.

The rest of the seeders work as expected, the price list factory has no country and no currency methods to create such data.

# Repositories and Services:

Although it is a small project but just for demonstration I’m using repositories and a service.

At the beginning I thought of creating an action or a service for the logic of the two required APIs. But when I added caching, I thought it might be a good idea to create a repository in which accessing the database is handled, and caching logic and exception handling done in the service.

This all was done to make the Controller cleaner and try to follow the Single Responsibility Principle (SRP). Also, the repository can be easily tested using unit tests.

The repository is used inside the service via Dependency Injection. This is done in the constructor of the service. The service is also injected into the controller inside the constructor.

# Controllers:

Since I’m using Web and API, I prefer to split the concerns. For the API, I used only one Controller inside the API Directory. The two other controllers are used in Web and are located in the Admin Directory.

# Views:

We only need views for the admin and they are added inside the Admin Directory.

Since the admins are allowed to define price lists and products, I used Laravel Breeze to benefit from its authentication.

There is one role so Authentication is enough in this situation, as for the users of the API, there is no need to authenticate them, therefore no need for a login process.

For the front-end I used Laravel UI and created CRUD operations for the products and price lists in which we define a price list item for the products.

# Rate Limiting:

Having our API open to anyone leads us to the need of rate limiting, to try to prevent any attacks on it.

In the boot method inside AppServiceProvider we provide a rate limiting rule, and is used in the API routes file in the middleware throttle:api.  
The rule permits for a number of requests per minute coming from the same IP Address.

# APIs:

## Get all products API:

I found this API really ambiguous and I had so many questions and cases in my head. However, I chose to comply with the following scenario:

The API returns all the products in the database, each one with the price list with the lowest priority which also contains the applicable price, if the price list has no country or currency, a default country code (‘US’) and currency code (‘USD’) are used, both are defined as constants in their models respectively.  
If a product has no applicable price list, it will default to the base price in the product, with the default country and currency.

The drawback and ambiguity of this is that the response now will contain the prices of the products with different currencies, which is for me actually unacceptable and needs more clarification.

A better use of the API would be to pass the same parameters of the single product API and get the price lists accordingly. But even this way, for products with no price lists it will create the currency problem again.

Another possible way is to get the IP of the user and use an API to get the country, this is demonstrated in the GetCountryCodeFromIpAction but not used. But even this way requires a connection between the country and the currency, in addition to assigning a default currency for each country and a default date as the current date, this way we get all the parameters we want and return the applicable price with currency.

The query which gets all the products is found in the ProductRepository. It is using Laravel Query Builder and raw SQL. For large datasets, query builder outperforms Laravel Eloquent ORM even with the use of eager loading and the correct relationships defined in the models.

When using this API, the result is cached with the cache key defined in the controller. The key contains a prefix used to catch the key elsewhere to remove the cash.  
Cache removal (forgetting) is done in the admin part, where the admin performs any operation on the product or the price list, it uses the prefix const defined in the model to catch the cache and remove it.

The API also uses Laravel Pagination. In such APIs, backend pagination is essential for query performances.  
Even the admin should have its own pagination for the lists (products and price lists), but for simplicity I stuck to returning all the data in the model.

The API also receives the order parameter, which sorts the products by price in the desired way.

## Get single product API:

This API has a specific request to validate the parameters provided by the user and return the appropriate response.  
Applying requests can also be applied in the admin section, but it is not done for simplicity.

In the service, the country and currency codes are validated against our database, an additional reason to do this is when there is no country or currency returned with the response (the price list has no country or currency and it applies to the parameters).

# Testing Using PEST:

There is a ProductTest file in the tests/Feature Directory. This file contains 2 test functions to test the result of the repository methods for each of the two APIs. So, it basically creates a scenario for each of the APIs with the correct data and get the results as expected.

# Postman:

A Postman Collection is provided at the root of the project for the 2 APIs.