19.02.2025, Wrocław

Contractor: Paweł Bączkiewicz

Travelplanet – Interview Task

Project Setup:
- Laravel 11.43.0
- PHP 8.3 / 8.4
- database/sqlite
- docker

Project Structure:

- **src/Module/PaymentScheduleAPI**
  - **Application**
    - DTOs
    - Validators
  - **Domain**
    - Entities
    - Rules
    - Services
  - **Infrastructure**
    - Middleware
    - Services
  - **Presentation**
    - Http
      - Controllers

The module provides an endpoint for generating payment schedules using a RESTful API.
It supports versioning and allows for the development of new payment plans based on product parameters.

- **src/Module/ProductManagement**
    - **Application**
        - **Commands**
            - CreateProduct
            - DeleteProduct
            - UpdateProduct
        - DTOs
        - **Queries**
            - GetAllProducts
            - GetProduct
        - Services
        - Validators
    - **Domain**
        - Entities
        - Repositories
    - **Infrastructure**
        - Api
        - Bus
        - Persistence
        - Providers
    - **Presentation**
        - Http
            - Controllers

The module provides a simple endpoint for managing products and communicates via API with the payment plan generation module using a RESTful API.

- **src/Module/Shared**
    - **Application**
        - DTOs
        - Validators
    - **Domain**
        - Entities
        - Service
        - ValueObjects
    - **Infrastructure**
        - Facades
        - Middleware
        - Providers
        - **Services**
            - Logger

The module contains objects within a DDD structure used by both endpoints, as the project follows a Modular Monolith architecture.

Compliance with project requirements:

Project Setup:
• Use PHP (preferably PHP 8.1 or later).
• Use a framework of your choice (e.g., Laravel, Symfony).
• Use Docker container to run the project.

As I mentioned at the beginning, all the above requirements have been met.

Application Functionality:
• Create an application that calculates a payment schedule based on the provided product information.
• The product information should include:
    o productType: Products are grouped by Type.
    o productName: Name of the product.
    o productPrice: Object containing amount and currency.
    o productSoldDate: Date when the product was sold.

1. The task does not specify exactly how the product should be delivered, but I assumed that a simple product CRUD would be a good approach → ProductManagement.
2. Payment rule calculation via RESTful API → PaymentScheduleAPI.

Product as a Domain Entity/Aggregate → *ProductManagement\Domain\Entities\Product.php.*
ProductType is a ValueObject/enum → *Shared\Domain\ValueObjects\ProductType.php.*
ProductPrice is a ValueObject → *Shared\Domain\ValueObjects\Money.php.*
Money encapsulates part of the functionality of Moneyphp.

API Endpoints:
• Define the RESTful API for Payment Schedule Calculation
• Describe versioning approach for future versions of API with Breaking Changes

API versioning for future use is implemented through routing - the version number in the URL determines which version will be applied. In my example, the new „v2” version introduces a new type of payment schedule for the "Student" product type. A request with the same product to version „v1” will trigger the Fallback rule (with the exception of the June rule, which has higher priority).

PaymentScheduleAPI\Infrastructure\Services\ApiVersionService.php
PaymentScheduleAPI\Infrastructure\Middleware\ApiVersionMiddleware.php

---

Payment Schedule Calculation:
• The payment schedule should be returned as JSON which contains:
    o amount: Payment amount.
    o currency: Currency of the payment.
    o dueDate: Due date for the payment.

The API provides payment plans in the specified JSON format as an array of one or more installments, additionally wrapping the response with other useful information.

PaymentInstallment – a ValueObject, serialized according to the business requirements standard.

Shared\Domain\ValueObjects\PaymentInstallment.php
PaymentScheduleAPI\Application\DTOs\ResponsePaymentScheduleDTO.php
PaymentScheduleAPI\Application\DTOs\ErrorDTO.php

• Implement rules for calculating the payment schedule based on productType and productSoldDate. For example:
• If the product is sold in June, the payment schedule is calculated into 2 installments where the 1st installment is 30% paid immediately and the rest is paid 3 months from the end of the current month.
• Prepare at least 3 different rules plus 1 fallback rule to be used whenever there is no other suitable rule.

PaymentScheduleAPI\Domain\Rules\PaymentRuleInterface.php
PaymentScheduleAPI\Domain\Rules\JunePaymentRule.php
PaymentScheduleAPI\Domain\Rules\PremiumPaymentRule.php
PaymentScheduleAPI\Domain\Rules\StandardPaymentRule.php
PaymentScheduleAPI\Domain\Rules\StudentPaymentRule.php
PaymentScheduleAPI\Domain\Rules\FallbackPaymentRule.php

PaymentScheduleAPI\Domain\Services\PaymentScheduleCalculatorInterface.php
PaymentScheduleAPI\Domain\Services\PaymentScheduleCalculator.php

• Rules can be defined in different currencies, but the response has to be in the same currency as request

Here, I encountered difficulty with the interpretation. The task specifies a percentage rule for a product sold in June. Percentage-based rules seem to make sense in this case, so following this reasoning, I applied other rules as percentage-based as well. For amount-based or mixed rules, the business logic would be much more complicated.

Additionally, it is stated in the assumption above that the response should be in the same currency as the request, meaning that the payment installments returned by the API should be in the same currency as the product sold. Given these two factors, one might consider that the currency in the payment rules is unnecessary, as currency is transparent in percentage-based calculations.

However, perhaps the task author meant to establish future installment currencies - if that were the case, currency conversion would be necessary. One could imagine a situation where the product is sold in PLN, but the installments are calculated in EUR on the product sale date. In that case, the customer could see an amount in PLN, even though the payment is in EUR, but a different value would be displayed every day to be paid in PLN. It seems like a strange behaviour.

This part would need clarification. I decided not to consider the currency. However, for the future, I added a conversion object and showed where this conversion could occur.

Shared\Domain\Service\CurrencyConverter.php
PaymentScheduleAPI\Domain\Rules\FallbackPaymentRule.php

• Requests can come with time in different timezone

I decided that the date format for the PaymentScheduleAPI endpoint should comply with the ISO8601 standard, while the ProductManagement module will convert the date to UTC and store it in the database in that format. The issue remains on the frontend, where dates should be displayed according to the user's locale. Therefore, I implemented simple JavaScript scripts that handle all dates on the frontend and convert them from the UTC format to the user's timezone.

Shared\Application\Validators\ProductRequestValidatorAbstract.php
Shared\Application\Validators\SimpleProductRequestValidatorAbstract.php
ProductManagement\Infrastructure\Persistence\EloquentProductRepository.php

Frontend Javascript scripts for managing timezone:
resources\views\components\layout.blade.php
• Payment schedule rules has to be immutable – final readonly, only getters

Architecture requirements
• Use CQRS pattern to separate functional parts of application

ProductManagement\Application\Commands\CreateProduct\CreateProductCommand.php
ProductManagement\Application\Commands\CreateProduct\CreateProductCommandHandler.php
ProductManagement\Application\Commands\DeleteProduct\DeleteProductCommand.php
ProductManagement\Application\Commands\DeleteProduct\DeleteProductCommandHandler.php
ProductManagement\Application\Commands\UpdateProduct\UpdateProductCommand.php
ProductManagement\Application\Commands\UpdateProduct\UpdateProductCommandHandler.php

ProductManagement\Application\Queries\GetAllProducts\GetAllProductsQuery.php
ProductManagement\Application\Queries\GetAllProducts\GetAllProductsQueryHandler.php
ProductManagement\Application\Queries\GetProduct\GetProductQuery.php
ProductManagement\Application\Queries\GetProduct\GetProductQueryHandler.php

ProductManagement\Infrastructure\Bus\CommandBus.php
ProductManagement\Infrastructure\Bus\QueryBus.php

• Implement Strategy pattern for different payment rules

PaymentScheduleAPI\Domain\Services\PaymentScheduleCalculatorInterface.php
PaymentScheduleAPI\Domain\Services\PaymentScheduleCalculator.php

• Use proper Value Object for manipulation with Money

Shared\Domain\ValueObjects\Money.php

Data Storage:
• Use a database of your choice (e.g., MySQL, PostgreSQL, SQLite) to store product information and payment schedules.

app\Models\Product.php
app\Models\PaymentInstallment.php

database\migrations\2025_02_15_201306_create_products_table.php
database\migrations\2025_02_15_201358_create_payment_installments_table.php

database\factories\ProductFactory.php

• Ensure the database schema is well-designed and normalized.

- UUID as ID
- ProductType as enum
- Currency as enum
- PriceAmount as Decimal(10,2) – no rounding

Enums are not supported by SQLite (converted into string) but it is supported by i.e MariaDB
I considered to store values in database as integers like Moneyphp do and convert them into decimals.

• Use DB Transactions

ProductManagement\Infrastructure\Persistence\EloquentProductRepository.php

• Implement Repository pattern

ProductManagement\Infrastructure\Persistence\EloquentProductRepository.php
ProductManagement\Domain\Repositories\ProductRepositoryMapperInterface.php
ProductManagement\Domain\Repositories\ProductRepositoryInterface.php
ProductManagement\Infrastructure\Persistence\EloquentProductRepositoryMapper.php

Authentication:
• Implement basic authentication (e.g., API key, token-based authentication).
• Use authorization for more different user roles
• Secure app in accordance to OWASP top 10

I used Laravel/sanctum for simple API token-based authentication.

PaymentScheduleAPI\Presentation\Http\Controllers\AuthController.php
PaymentScheduleAPI\Infrastructure\Middleware\RefreshTokenMiddleware.php
ProductManagement\Infrastructure\Api\ApiClient.php
database\migrations\2025_02_16_112203_create_personal_access_tokens_table.php

I defined couple roles for different users and presented some simple way to access some for some parts.

Shared\Domain\ValueObjects\RoleEnum.php
database\migrations\0001_01_01_000000_create_users_table.php

OWASP top 10 is mostly provided with Laravel

Performance, Logging, Monitoring
• Define expected response times

API timeout - *ProductManagement\Infrastructure\Api\ApiClient.php*

• Log requests in structured format

Would be good to have separated files in future for debug/warning/info/error…
I provided one file log → travelplanet.log

Shared\Infrastructure\Facades\Log.php
Shared\Infrastructure\Providers\LoggerServiceProvider.php
Shared\Infrastructure\Middleware\LogHttpErrors.php
Shared\Infrastructure\Services\Logger\ApplicationLogger.php

Presented how logging should be done here:

ProductManagement\Infrastructure\Bus\CommandBus.php
ProductManagement\Infrastructure\Bus\QueryBus.php

• Log application errors in more levels by severity

src\Modules\Shared\Infrastructure\Services\Logger\ApplicationLogger.php

• Collect data for metrics needed to define SLOs

only simple → Shared\Infrastructure\Middleware\ResponseTimeMiddleware.php

• Specify monitoring endpoints (health check, readiness)

not provided

Error Handling:
• Implement proper error handling and return appropriate HTTP status codes.

How should be done presented here:
ProductManagement\Presentation\Http\Controllers\ProductController.php

Documentation:
• Provide API documentation (using Swagger/OpenAPI).

not provided

• Include instructions on how to set up and run the application.

README.md

Testing:
• Write unit tests, integration tests and API contract tests for your application (e.g., using PHPUnit, Behat).

27 tests:
- 12 unit
- 14 functional
- 1 integration

What should be done next (if I have more time)

1. Check code with PHPStan
2. Check code with PHPCS
3. Prepare more tests covering all parts
4. Refactor code (especially API/Logging/ErrorHandling/ExpectedTimeResponses ...)
5. Provide what was not provided or partially provided:
a) metrics for SLOs
b) provide API documentation
c) specify monitoring endpoints (health check, readiness)

Thank you for this oppurtunity.
Paweł.