saas.group Tower Test Task #1

Greg Muszynski (g.muszynski.12@aberdeen.ac.uk)

Overview

This is a supporting document for the simple proof-of-concept subscription management app built with Ruby on Rails 7 using PostgreSQL as a solution to the question outlined in Task #1. While I made sure to carefully comment all the notable points in the source code, I also thought that it would still be useful to have a supporting write-up.

The app allows creating subscription plans, subscribing to those plans with a specified number of seats, and applying coupons to generate discounts. It also connects to an external payment provider via a simple API to update subscription prices whenever changes occur, e.g. when a coupon is applied or removed.

Main Entities and Features

Plans Define subscription tiers with unique titles and unit prices.

Subscriptions Users can subscribe to a plan, specifying the number of seats. Each subscription has an auto-generated UUID as an external ID for integration with the payment provider.

Coupons Offer percentage-based discounts on subscriptions. Coupons have a maximum number of uses (charges) and track how many times they've been applied.

Discount Logic Applying a coupon reduces the subscription's `effective_price` while keeping the original `unit_price` intact for reference. Removing a coupon restores the `effective_price` to the original `unit_price`.

Coupon Immutability Once a coupon is applied to a subscription, it cannot be edited or deleted.

Payment Provider Integration The app communicates price changes (effective price) to the payment provider via a POST request to `/subscriptions/:external_id` with the updated `unit_price`. The code responsible for this is encapsulated in `PaymentProviderNotificationService`.

Key Assumptions

- The payment provider accepts POST requests to `/subscriptions/:external_id` with a JSON body containing the updated `unit_price` (effective price).
- Each subscription has a unique UUID (`external_id`), generated automatically using PostgreSQL's `gen_random_uuid()`.
- Coupons can be applied to multiple subscriptions up to their `max_charges` limit.
- Prices are stored as decimals with precision 10 and scale 2 to handle monetary values accurately.
- PostgreSQL is used, leveraging its decimal type for prices and uuid type for external IDs.

Setup

In my own setup I use `rbenv` for Ruby version management and for this app I used Ruby version 3.4.2 (declared in `.ruby-version`).

```
git clone https://github.com/greg-asc/TowerSubscriptionCouponDiscounts.git

cd TowerSubscriptionCouponDiscounts

rbenv install

bundle install

bundle exec rails db:setup
```

Note: I chose PostgreSQL for this implementation and created the database role `tower` with password `S3cr3tP455w0rd` (everything is included in `config/database.yml`).

Usage

The best way to interact with this app is probably via the console:

```
bundle exec rails c
```

Here's a simple demo you can try:

```
irb(main):001> plan = Plan.create title: "Basic", unit_price: 19.99

irb(main):002> subscription = Subscription.create plan: plan, seats: 5, unit_price: plan.unit_price

irb(main):003> coupon = Coupon.create code: "SAVE10", percentage_discount: 10.00, max_charges: 100

irb(main):004> subscription.apply_coupon coupon

irb(main):005> subscription.remove_coupon
```

Testing

The app includes RSpec tests for all models, covering all of the key requirements outlined in the Task #1 description. It also demonstrates the assumptions made in places where the task description left some room for interpretation. Use the following command to run all tests with the full output:

```
bundle exec rspec -f d
```

API Integration

The app attempts to simulate an real-world integration with a payment provider to update subscription prices whenever the effective price changes, e.g. when a coupon is applied or removed. The integration uses HTTParty to make the POST request

`POST /subscriptions/:external_id  { "unit_price": <effective_price> }`

where `effective_price` is the price calculated with a discount if a coupon is applied, and the default `unit_price` otherwise. The subscription's `external_id` is an auto-generated UUID used to identify the subscription on the payment provider's side.

The tests use WebMock to intercept requests from `PaymentProviderNotificationService` and imitate a real API endpoint responding with HTTP 200 (see `spec/models/subscription_spec.rb:45` for example).

Key Implementation Details and Decisions

`unit_price` vs `effective_price` In order to make the code reasonably readable I decided to use `unit_price` to represent the pre-defined price of a subscription or plan and `effective_price` to represent the result of calculating any discounts on `unit_price`. The task description expects the JSON body of the API request to have the `unit_price` field, which I kept in my implementation to match the requirements, however, I found that internally it made more sense to me to allow accessing both the unit price and the discounted price, as opposed to perhaps overwriting the starting unit price.

`unit_price` - `Subscription` vs `Plan` Additionally, I was unable to fully understand the difference between `unit_price` in `Subscription` vs `Plan` so for the purpose of this task I simply assumed that it gets passed on from `Plan` to `Subscription` but I implemented everything is such a way that a more sophisticated behaviour could be supported if needed.

- Prices are stored as decimals to prevent precision issues common with floats.
- Subscriptions use UUIDs for `external_id` to ensure uniqueness and compatibility with external systems.

- Once a coupon is applied to a subscription, it becomes immutable to maintain data integrity. Once it's unapplied, editing is re-enabled.
- Price updates are sent from the notification service to the payment provider after successful database transactions.