Documentação:

Nesta documentação estarei descrevendo os principais pontos de minha implementação do teste. Gostaria de desde já pedir desculpas, pois na elaboração acabei “esticando muito a baladeira”, sobrando pouco tempo para fazer mais testes automatizados e trabalhar em mais recursos.

Como instalar a api.

- No terminal, dentro da pasta executar “docker-compose up”.
- Executar o script “finish-setup.sh” para fazer as configurações do composer
- Executar o comando “docker exec -it php-fpm php artisan migrate” para executar as migrations.
- Estão disponíveis 3 filas de execução, “csv”, “payment” e “emails”, para executar basta executar o comando: “docker exec -i -t php-fpm php artisan queue:work --queue=csv,payment,emails”
- Os arquivos de configuração “.env.example” e “.env.testing.example” estão disponíveis dentro da pasta “environments” já com os dados para acesso.

Documentação

- Na pasta “documentation” está o arquivo de coleções do postman “Themembers.postman_collection.json” que possui as principais requisições
- Também na pasta “documentation” está disponível o arquivo “diagram.png”, que evidencia o modelo de banco de dados.
- Na pasta “documentation/csv files”, temos 2 arquivos, que podem ser utilizados em uma rota para criar vários registros ao mesmo tempo.

Serviços

- Servidor WEB, aberto na porta 8080, simulando o servidor de aplicação
- Servidor de banco de dados, na porta 3306
- Servidor de email de testes, na porta 8025

Principais requisitos

- Crud completo de “bueyrs”
- Criar vários “buyers” via arquivo csv
- Crud completo de “products”
- Criar vários “products” via arquivo csv
- Criar um pedido de pagamento
- Verificar um pagamento
- Verificar boletos pagos

Crud de “buyers”

- GET /api/buyers
    - Lista de “buyers”
    - Traz a lista de “buyers” paginada do banco de dados
- POST /api/buyers
    - Criar um “buyer”
    - Cria um Buyer no banco de dados, através de dados da requisição
- GET /api/buyers/<buyer_id>

    - Mostra um buyer
    - Detalha os dados de um determinado “buyer”
- PATCH /api/buyers/<buyer_id>
    - Atualiza um buyer
    - Atualiza o registro de um “bueyer” através de dados da requisição
- DELETE /api/buyers/<buyer_id>
    - Exclui um “buyer”
    - Exclui o registro de banco de dados de um buyer
    - *** Não tive tempo hábil para implementar um recurso de soft delete ou de exclusão via cascade do recurso

Criar vários “buyers” via arquivo csv

- POST /api/buyers/post-csv
    - Cria vários “buyers”
    - Sobe um arquivo estruturado .csv com dados de arquivos, e cria vários “buyers” no banco de dados
    - Logo após subir o arquivo, a requisição já retorna sucesso
    - A criação de registro acontece de forma assíncrona, melhorando o desempenho da aplicação.
    - *** Não tive tempo hábil para aprimorar este recurso, o ideal seria validar os dados linha a linha, no fim da execução, haver um email de desenvolvedor que receberia detalhes da execução, com a quantidade de registros criados, com falha e a cópia do arquivo no anexo de email

Crud completo de “products”

- GET /api/products
    - Lista de “products”
    - Traz a lista de “products” paginada do banco de dados
- POST /api/products
    - Criar um “product”
    - Cria um “product” no banco de dados, através de dados da requisição
- GET /api/products/<product_id>
    - Mostra um “product”
    - Detalha os dados de um determinado “product”
- PATCH /api/products/<buyer_id>
    - Atualiza um product
    - Atualiza o registro de um “product” através de dados da requisição
- DELETE /api/product/<productr_id>
    - Exclui um “product”
    - Exclui o registro de banco de dados de um product
    - *** Não tive tempo hábil para implementar um recurso de soft delete ou de exclusão via cascade do recurso

Criar vários “products” via arquivo csv

- POST /api/products/post-csv
    - Cria vários “products”
    - Sobe um arquivo estruturado .csv com dados de arquivos, e cria vários “products” no banco de dados
    - Logo após subir o arquivo, a requisição já retorna sucesso
    - A criação de registro acontece de forma assíncrona, melhorando o desempenho da aplicação.
    - *** Não tive tempo hábil para aprimorar este recurso, o ideal seria validar os dados linha a linha, no fim da execução, haver um email de desenvolvedor que receberia detalhes da execução, com a quantidade de registros criados, com falha e a cópia do arquivo no anexo de email

Criar um pedido de pagamento

- POST /api/payments
    - Cria um pedido de pagamento
    - Vindo da requisição, o “document ” do “buyer”, “amount”, “payment” e uma lista de “products”
    - Funciona de forma assíncrona, após o pedido, um job faz o processamento de pagamento por baixo dos panos, deixando a aplicação de forma melhor performática
    - Cada um dos métodos depagamento funciona de uma forma
        - PIX: Uma simulação de acesso à api com um certo delay, logo após um número aleatório decide se foi processado com sucesso ou erro.
        - Cartão de crédito: Uma simulação de acesso à api com um certo delay, logo após um número aleatório decide se foi processado com sucesso ou erro.
        - Boleto: Um email é enviado com a rota de “payment document” (será explicado no próximo tópico)

Verificar um pagamento

- GET api/payments/<payment_id ou payment_hash>/document
    - Rota para verificar o status de pagamento
    - Cada um dos métodos de pagamento funciona de forma diferente
        - PIX: Um serviço de geração de qr code simula a geração de um token de pix. É mostrado logo abaixo a situação do pagamento, PS: acessar essa rota via navegador, assim consegue ver a simulação de delay do processamento do servidor.
        - Cartão de crédito: Uma página HTML que simula um comprovante de pagamento, PS: acessar essa rota via navegador, assim consegue ver a simulação de delay do processamento do servidor.
        - Boleto: Usei uma página que gera um boleto falso para gerar um boleto, que o usuário poderia baixar.
        -

Verificar boletos pagos

- Comando artisan do laravel “`payments:verify-boletos`”
    - Simula o acesso de uma api, retornando uma lista de hashs de pagamento já pagos, que são estados nos pagamentos da api como completos

Testes automatizados

- Sobre os tests, é importante eu comentar que acabei focando um pouco demais nas features de aplicação, e fiz poucos tests, relacionados somente aos cruds de “buyers” e “products”, porém já dá uma idéia de minha organização relacionado a fazer testes automatizados
- Outros testes que faria
    - Testes relacionados a paginação de cruds. Principalmente a pesquisa usando queries e order_by
    - Testes relacionados ao show, update e delete
    - Testes relacionados ao arquivo .CSV
    - Testes relacionados ao processamento de pagamentos
    - Testes relacionados ao envio de email
    - Testes relacionados a requisição de serviços que simula pix e boleto

Plus, alguns detalhes a mais que decidi colocar na aplicação

- Texto sempre personalizados, por região e idioma
    - Os textos foram traduzidos para portugês brasileiro, as mensagens de controladores estão preparados para receber outras traduções.
- Criação de mais interfaces, tornando o objeto laravel e sua estrutura independente do serviço
    - Com a criação da interface CanBePayd, é possível que fosse reutilizado em outros sistemas de forma simples (Fiz de forma simplória, não tive tempo de trabalhar melhor esta idéia).
- Email sempre como Queueable
    - Emails agora são enfileráveis, e possuem sua próprio fila de trabalho, organizando melhor o projeto e suas filas.