Design.md

Payments API Service

This document describes the design of the Payments API coding exercise.

Design Goals

- Flexible and verifiable REST API contract
- Full event log for all Payment changes
- Clear separation between persisted entities and request data, so that the API can be changed or swapped out completely

Design Overview

Straight forward CQRS/Event Sourcing design.

API

The API is fully generated from a Swagger definition file. We get the following for free:

- Request validation
- Structured response
- Verified contract
- Mapping of request data to application data models
- Client generation provided by Swagger

The specific API structure can be found in the `swagger.yaml` file. It will look very familiar, as it's based on the official Form3 `swagger.yaml`.

Command Layer

When we receive a data change request we build a domain command:

- **POST → CreatePayment**
- **PATCH → UpdatePayment**
- **DELETE → DeletePayment**

Commands are built from API requests and validated for correctness separately from the API parameter validation.

Note: Only a very tiny subset of attributes are validated on the command, mostly for demonstration. I didn't want to spend time describing validation for dozens of attributes.

Payment Aggregate

Extremely simple. Only tracks the state of the Payment - **Created**, or **Deleted**.

Projection

Separate read model that is built from the domain events we receive. Can be fully rebuilt from event data. Each change to a Payment automatically updates it's `Version`

Design Tradeoffs

Data Type conversions

The complete and explicit separation between all the layers of the system provides great flexibility, but comes at the cost of tedious conversion between all the layers.

`API Request Data → Command Data → Event Data → Read Model`

`API Request Data → Repository Filter → Persisted Entity → API response model`

Each of those is a separate entity that can have it's own structure. Type conversion code is mostly development time cost.

Workaround: For the purpouse of the test having hundreds of lines of static type conversion would be extremely tedious and irrelevant. Everything currently serialises to the same JSON and type conversions are done by Marshalling/Unmarshalling JSON.

The `Update` from CRUD maps poorly to Event Sourced design

The domain event of "Update Attributes" is meaningless for the most part. Ideally the `PATCH` HTTP request is deconstructed in meaningful domain commands like `ChangeAmount` , `ChangeCurrency` , etc. Something that reflects a real domain action. `Update` is a data operation, not a domain one.

Technologies

- Go - just wanted to try it out. This is the first thing I've ever written in go.
- EventHorizon - CQRS/ES framework for Golang. Chosen just because it seemed popular.
- Swagger - Generating the API server and supporting HTTP API code.
- MongoDB - just because it is EventHorizon's default storage. After working with MongoDB for the past 7 years I would never use it for anything that needs to reliably store data. I just default to Postgresql for everything as it's extremely reliable and predictable.

An Alternative

I generally avoid REST APIs for domain services and prefer gRPC as an API. Introducing an API frontend(or lots of them) that can provide public APIs - be it a very stable one for integrations or a frequently changing one for javascript frontends.

However, this seemed way out of scope for this exercise.

Personal Notes

While this submission is a month late. The test took me 4 days - one weekend and two weekday evenings. This is actually the first ever code I've written in Go, may be a bit rough around the edges as I'm still learning what is "idiomatic go". One example of this is my use of BDD tesing, instead of the standard go test table approach