Payments API Design

Payments API provides a simple RESTful Payments API microservice based on a HTTP server implemented in Go (and the GoKit toolkit for microservices) that deals with Json format files to provide CRUD functionality against a Postgresql database.

I have used Gokit to help in separating implementation concerns by employing an onion layered model, where at the very core we have our use cases or business domain (source code dependencies can only point inward) and then wrapping that with other functionality layers.

I have chosen Postgresql as the database backend because, since this is a fintech domain application, reliability and data integrity are key and often take precedence before the high performance, sharding and scalability that a noSql DB solution would offer (guarantees ACID operations).

I have used GORM (a Go ORM) to benefit from having a rich, object oriented business model and still be able to store it and write effective queries quickly against a relational database.

Through GORM I have opted for a “soft delete”, meaning that when deleting a payment it is not actually deleted from the database, it is just marked in the “DeletedAt” column with a timestamp. Only records with an unpopulated “DeletedAt” section are returned in this API's GET operations.

Payment Identifiers are Universal Unique Identifiers because providing successive identifiers (such as 0,1,2,3...etc) would be a security risk (if an attacker would know a successive identifier, he would then be able to tell and target neighboring payments and would have more information than it is actually needed)

The implemented prototype needs to be able to:

- Fetch a single existing payment resource based upon a provided payment ID
- Create, update and delete a payment resource
- List a collection of payment resources (here is what a list of payments might look like)
- Persist resource state (to a database).

The implementation roughly consists of a HTTP server and router that deals with the various HTTP requests and responses and performs the routing to and from the various endpoints of the API. There is also builtin functionality to analyze arguments that are being passed in from the command line (such as HTTP server port and the path to the postgres.toml database configuration file) but also to parse the Postgres DB configuration file and start a database connection. The Gokit wrapper module, is used in order to implement a decorator pattern where concerns such as logging, monitoring, transport, circuit breaking are separated and do not introduce any dependencies in the core functionality code (the actual Payments Service functionality).

The core Payments Service functionality is where the business logic resides. It leverages GORM

and a database model to implement typical CRUD functionality against the Postgres database layer.

The diagram in fig.1 show the architecture of the system and how the various parts come together to form the Payments API prototype.

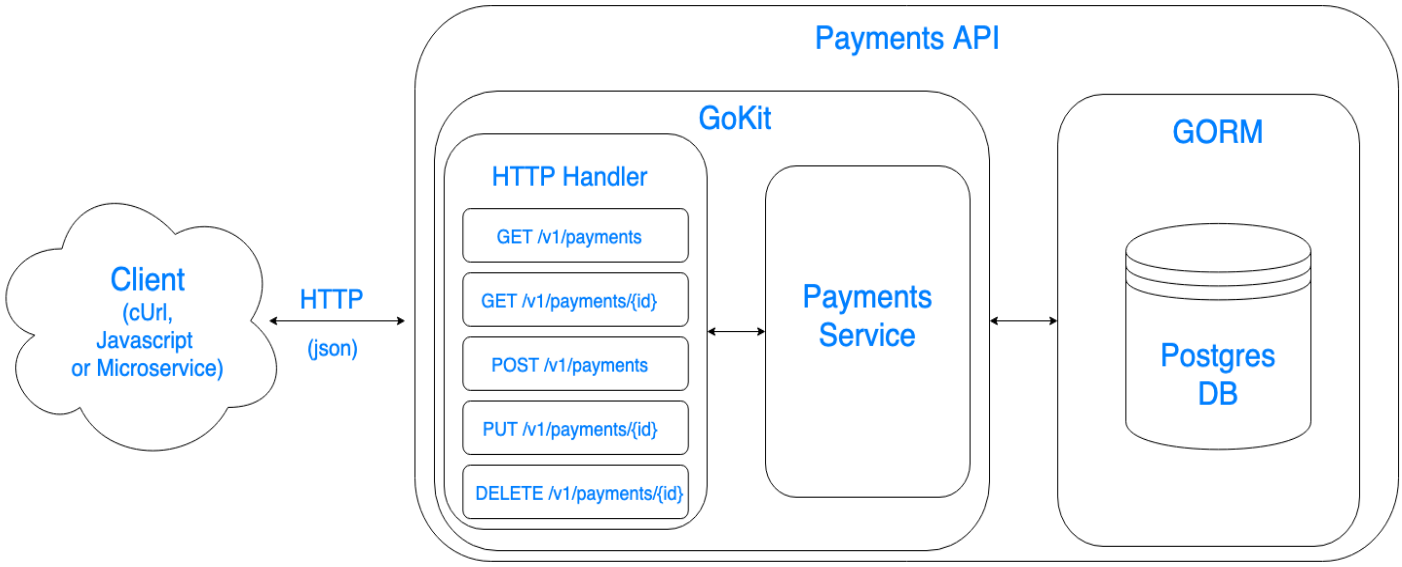

Fig.1 Payments API architecture

The model for how the database is structured can be visualized in fig.2

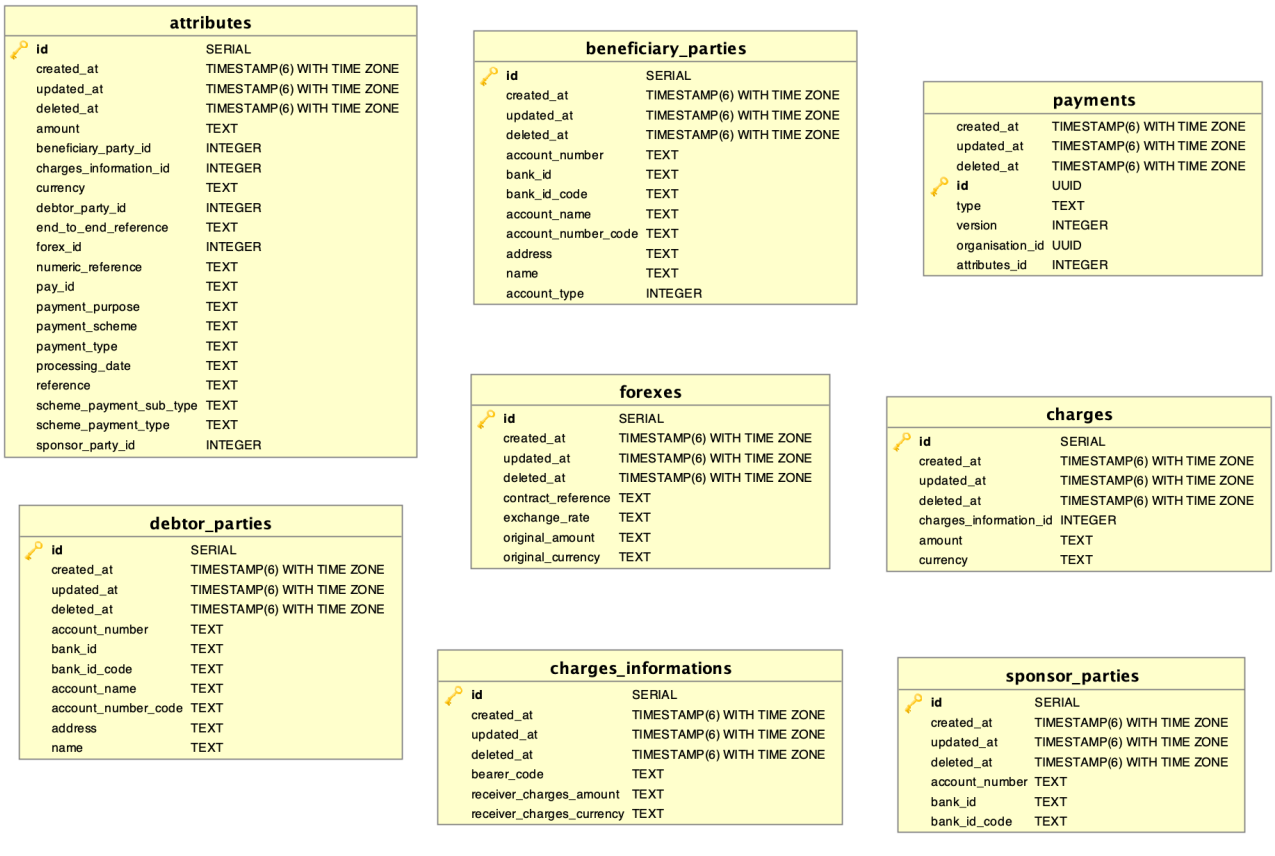

Fig.2 Payments API database model

The Payments API description is summarized in the table.1:

Verb	Endpoint	Content	Description	Response
GET	/v1/payments	No Body / No Payload	Lists all available recorded payments	**Success:** **Code:** 200 (<json formated list of payments>) **Fail:** **Code**: 500 (err: Could not GET list payments)
GET	/v1/payments/{id}	No Body / No Payload	Lists a specific payment based on provided id	**Success**: **Code**: 200 (<*json formated payment*>) **Fail**: **Code**: 500 (err: Could not GET payment record not found)
POST	/v1/payments	Json Payload (ex. payment0.json)	Creates a new payment based on the Json file provided as payload and generates a unique payment ID as a response	**Success**: **Code**: 201 ({"created_id":"<*payment_uuid*>}) Fail: **Code**: 500 ({"err: Could not Create(POST) payment"})
PUT	/v1/payments/{id}	Json Payload (ex. payment1.json)	Updates an existing payment based on the provided identifier (id) and on the json file provided as payload	**Success**: **Code**: 201 ({"updated_id":"<*provided_payment_ uuid*>}) Fail: **Code**: 500 (err: Could not Update(PUT) payment)
DELETE	/v1/payments/{id}	No Body / No Payload	Deletes (soft deletes) a payment entry from the database based on the provided id. Basically, marks the DeleteAt cell with a timestamp and the entry is no longer taken into account by any GET operation.	**Success**: **Code**: 200 ({"DeletedAt":<*timestamp_of_deltetion*>) **Fail**: Code: 500 (Could not DELETE payment record not found)

Table.1 API Desing