Payments API Design Document

Author: Wojciech Gac

Overview

The following documents describes a RESTful API solution for managing payments. The API operates on the concept of entity (a tangible representation of a single payment) and a set of operations on that entity, available to the user. The operations follow the standard pattern of CRUD (Create, Read, Update and Delete). In addition, there is also the aggregate view, suitable for listing all payments known to the system. Handling of the HTTP layer (requests, dispatching, return codes, etc.) has been delegated to the open-source library *mux* (available at: `github.com/gorilla/mux`).

Data Model

The central data structure of the payment model is the *Payment* structure, whose definition in Go is presented below:

```go
type Payment struct {
	Type           string        `bson:"type" json:"type"`
	ID             bson.ObjectId `bson:"_id" json:"id"`
	Version        int           `bson:"version" json:"version"`
	OrganisationID string        `bson:"organisation_id" json:"organisation_id"`
	Attributes     Attributes    `bson:"attributes" json:"attributes"`
}
```

Figure 1: Payment data structure

As can be seen, the Go tagging feature has been used to establish correspondence between structure fields and both JSON and BSON (as required by MongoDB, viz. Persistence Layer) field names. This will help considerably both with serializing/deserializing the data structures for use in HTTP requests and with translating the structures to comply with the requirements of MongoDB. The *Payment* structure includes a reference to another structure – *Attributes*, described below. Splitting the entity description into multiple nested structures increases modularity and readability of the model.

```
type Attributes struct {
	Amount               string               `bson:"amount" json:"amount"`
	BeneficiaryParty     Party                `bson:"beneficiary_party" json:"beneficiary_party"`
	ChargesInformation   ChargesInformation   `bson:"charges_information" json:"charges_information"`
	Currency             string               `bson:"currency" json:"currency"`
	DebtorParty          Party                `bson:"debtor_party" json:"debtor_party"`
	EndToEndReference    string               `bson:"end_to_end_reference" json:"end_to_end_reference"`
	Fx                   Fx                   `bson:"fx" json:"fx"`
	NumericReference     string               `bson:"numeric_reference" json:"numeric_reference"`
	PaymentID            string               `bson:"payment_id" json:"payment_id"`
	PaymentPurpose       string               `bson:"payment_purpose" json:"payment_purpose"`
	PaymentScheme        string               `bson:"payment_scheme" json:"payment_scheme"`
	PaymentType          string               `bson:"payment_type" json:"payment_type"`
	ProcessingDate       string               `bson:"processing_date" json:"processing_date"`
	Reference            string               `bson:"reference" json:"reference"`
	SchemePaymentSubType string               `bson:"scheme_payment_sub_type" json:"scheme_payment_sub_type"`
	SchemePaymentType    string               `bson:"scheme_payment_type" json:"scheme_payment_type"`
	SponsorParty         Party                `bson:"sponsor_party" json:"sponsor_party"`
}
```

Figure 2: Attributes data structure

Below are the remaining data structures, serving similar goals of modularity.

```
type Party struct {
	AccountName       string `bson:"account_name" json:"account_name"`
	AccountNumber     string `bson:"account_number" json:"account_number"`
	AccountNumberCode string `bson:"account_number_code" json:"account_number_code"`
	AccountType       int    `bson:"account_type" json:"account_type"`
	Address           string `bson:"address" json:"address"`
	BankID            string `bson:"bank_id" json:"bank_id"`
	BankIDCode        string `bson:"bank_id_code" json:"bank_id_code"`
	Name              string `bson:"name" json:"name"`
}
```

Figure 3: Party data structure

```
type Fx struct {
	ContractReference string `bson:"contract_reference" json:"contract_reference"`
	ExchangeRate      string `bson:"exchange_rate" json:"exchange_rate"`
	OriginalAmount    string `bson:"original_amount" json:"original_amount"`
	OriginalCurrency  string `bson:"original_currency" json:"original_currency"`
}
```

Figure 4: Fx data structure

```
type ChargesInformation struct {
	BearerCode              string   `bson:"bearer_code" json:"bearer_code"`
	SenderCharges           []Charge `bson:"sender_charges" json:"sender_charges"`
	ReceiverChargesAmount   string   `bson:"receiver_charges_amount" json:"receiver_charges_amount"`
	ReceiverChargesCurrency string   `bson:"receiver_charges_currency" json:"receiver_charges_currency"`
}
```

Figure 5: ChargesInformation data structure

```
type Charge struct {
	Amount  string `bson:"amount" json:"amount"`
	Curency string `bson:"currency" json:"currency"`
}
```

Figure 6: Charge data structure

Persistence Layer

MongoDB has been chosen as persistence layer for the API, chiefly due to the nature of typical usage scenario, i.e. whole-document retrieval or bulk retrieval, for which it is particularly well-suited. In addition, its schema-less character simplifies future modifications to the data model, as it only needs to be done in one place. The API uses the community-developed fork of the *mgo* library for integration with MongoDB (available at: `github.com/globalsign/mgo`). The abstraction layer over MongoDB operations is relatively thin, consisting mostly of calls to the *mgo* library and error handling.

Testing

The approach taken with respect to testing the API has been that of going through the entire life cycle of a payment entity, i.e. creation, retrieval, modification and deletion. The test suite used to that end starts by creating a test-only database in MongoDB and carrying out all the modifications there to avoid collisions with user-provided data.