Form3 Payment Server Specification

Author: Phil Muldoon

Date: 12th November 2018.

Revision: 3

Introduction

The payment server application was sponsored as part of an API challenge from Form3.

Application API

The intended API specification for this product is defined below:

- Create a new, single payment record in response to a valid POST request at the /payment/ URL,
- Update a single payment record in response to a valid PUT request at the /payment/{id} URL,
- Delete a single payment record in response to a valid DELETE request at the /payment/{id} URL,
- Fetch a single payment record in response to a valid GET request at the /payment/{id} URL, and
- Fetch a collection of payment records in response to a valid GET request at the /payments/ URL.

As requested in the requirements document, this API satisfies the RESTFUL API ideology. The data structure intended to be used to marshal the data back and forth from JSON is contained in Appendix A.

The API "https://api.test.form3.tech/v1/payments" is not accessible at present and so, for the purposes of this specification, the following conditions and terms are defined.

- A payment record is defined as a single, whole, coherent payment data structure as defined in Appendix A.
- A collection of payment records is any such number of records emitted by the /payments/ URL. It can have no records or an undefined upper bound of records.

- The primary index key for a payment record is defined as the ID field in the uppermost section of the payment record. This field is unique and indexed.
- Therefore only payment records with a unique, present ID field will be processed. Any payment record sent to the server that does not have an ID field, does not have data in the ID field, or contains data in the ID field that matches records already stored the backing store will be rejected.
- All other data, for the purposes of this specification, and lacking any further instructions in the requirements document, is defined as optional and can be either defined or blank.

Bar two exceptions, each of the APIs defined above will take one single payment record, encoded in JSON, as data. The exceptions are:

- The /payments/ GET request will take no parameters and return a collection in encoded JSON.
- The GET and DELETE APIs responding to /payment/ URL need only have the ID defined as the identifier for the information to be respectively processed. All other fields will be ignored.

The POST and PUT APIs responding to the /payment/ URL require the whole payment record, encoded in JSON, be posted along with the API call.

Technologies

All of the technologies utilised in the project will be open source. This will lower maintenance, lower development costs, and provide a view to code audits as and when they are appropriate.

The application will be written in the Go computer language (https://golang.org/).

The backing data store will utilise the MongoDB database technology (https://www.mongodb.com/what-is-mongodb). This is a scalable, high availability, horizontal scaling, and geographic distribution database that is open source. It stores documents in a binary JSON form (known as BSON) that makes data transition from client→server flexible and simple.

The data driver (that forms the bridge between the database and the application) will be mgo (https://labix.org/mgo). This is a rich, well established API for the MongoDB application.

The data transfer protocol for the database will be bson (https://gopkg.in/mgo.v2/bson). This closely mirrors JSON, the internet transfer protocol, and will not require translations to other languages for data store insertion (such as SQL databases).

The project will also use the Gorilla MUX router (http://www.gorillatoolkit.org/pkg/mux), a request router and dispatcher, to simplify the interface to the web.

Standard GO Unit tests will be utilised for testing as well as frameworks that encourage and allow behaviour-driven and test-driven development. The GO testing interface provides adequate framework for test-driven development but to enable behaviour-drive development the project will use GoConvey (http://goconvey.co/). This framework uses a Convey/So convention that allows rich user story interaction in the test framework.

The structure of the project will be as follows:

payments_server/

    server.go (dispatcher)

    model.go (database integrity and interaction)

    main.go (the main application)

    main_test.go (BDD and TDD testsuite)

Appendix A

```
type Payment struct {
      Type           string `json:"type"`
      ID             string `json:"id"`
      Version        int    `json:"version"`
      OrganisationID string `json:"organisation_id"`
      Attributes     struct {
            Amount           string `json:"amount"`
            BeneficiaryParty struct {
                  AccountName       string `json:"account_name"`
                  AccountNumber     string `json:"account_number"`
                  AccountNumberCode string `json:"account_number_code"`
                  AccountType       int    `json:"account_type"`
                  Address           string `json:"address"`
                  BankID            string `json:"bank_id"`
                  BankIDCode        string `json:"bank_id_code"`
                  Name              string `json:"name"`
            } `json:"beneficiary_party"`
            ChargesInformation struct {
                  BearerCode    string `json:"bearer_code"`
                  SenderCharges []struct {
                        Amount   string `json:"amount"`
                        Currency string `json:"currency"`
                  } `json:"sender_charges"`
                  ReceiverChargesAmount   string
`json:"receiver_charges_amount"`
                  ReceiverChargesCurrency string
`json:"receiver_charges_currency"`
            } `json:"charges_information"`
            Currency    string `json:"currency"`
            DebtorParty struct {
                  AccountName       string `json:"account_name"`
                  AccountNumber     string `json:"account_number"`
                  AccountNumberCode string `json:"account_number_code"`
                  Address           string `json:"address"`
                  BankID            string `json:"bank_id"`
                  BankIDCode        string `json:"bank_id_code"`
                  Name              string `json:"name"`
            } `json:"debtor_party"`
            EndToEndReference string `json:"end_to_end_reference"`
            Fx                struct {
                  ContractReference string `json:"contract_reference"`
                  ExchangeRate      string `json:"exchange_rate"`
                  OriginalAmount    string `json:"original_amount"`
                  OriginalCurrency  string `json:"original_currency"`
            } `json:"fx"`
            NumericReference     string `json:"numeric_reference"`
            PaymentID            string `json:"payment_id"`
            PaymentPurpose       string `json:"payment_purpose"`
            PaymentScheme        string `json:"payment_scheme"`
            PaymentType          string `json:"payment_type"`
            ProcessingDate       string `json:"processing_date"`
            Reference            string `json:"reference"`
            SchemePaymentSubType string `json:"scheme_payment_sub_type"`
            SchemePaymentType    string `json:"scheme_payment_type"`
            SponsorParty         struct {
                  AccountNumber string `json:"account_number"`
                  BankID        string `json:"bank_id"`
                  BankIDCode    string `json:"bank_id_code"`
            } `json:"sponsor_party"`
      } `json:"attributes"
```