# Design document

## Goal

The project is an API to process payments. The goal is to be able to have CRUD operations on payments: * Read a payment resource * Create a payment resource * Update a payment resource * Delete a payment resource * List payment resources

The payment resources should be persisted for later use.

We expect the API to be RESTFUL, and it might be consumed by JavaScript

## Assumptions

The proposed solution is based an this assumptions

- The project will be a RESTFUL JSON over HTTP API
- As we are dealing with payments, we make the assumption that the user will prefer to wait for an answer from the service rather than having to fetch the state of his action

## Proposed solution

### Architecture

The API will be composed of 3 main parts: * the routing * the handlers * the storage

The routing will be responsible to route the HTTP requests to the correct function. The handlers will handle one route. So we will have 6 handlers, see below for the routes. The storage will be responsible to store, retrieve the payments from the persistent storage

### Frameworks

We will use gorilla mux for the routing. We will use the std lib of go for the handlers. We will also use the std lib of go for the storage.

We will use the std lib of go and testify for the tests.

### Other notes

As the API might be consumed by a Javascript client, we will handle CORS pre-flight requests, with gorilla handlers

We do not have a requirement for authentication, so no auth mechanism will added.

The API requires to persist its state. We will use a simple, yet robust, storage: the file system. We will enforce a `fsync` on the disk to be sure the payments are persisted before returning an answer to clients. This way they will know if their request was successful or not.

We will also implement a validation on the incoming data: * make sure it is a valid JSON * make sure it respects the schema found in the annexe

## Proposed HTTP paths

To be able to change the API, it will be versioned, each HTTP path will be preceded by a number representing the version of the API. Here we will use `/1`

- Read a payment: `GET /1/payments/${uuidv4}`
  - Request parameters:
    - `uuidv1`: will represent the `id` of a payment
  - Responses:
    - `200`: the response body will be formatted as JSON. The payment attribute will be a payment, as one from the examples. `json { "payment": { ... } }`
    - `400` if `uuidv4` is not a valid UUIDv4, see
    - `404` if the payment id `uuidv4` doesn't exist in the storage
    - `500` for a server error, see
- Create a payment: `POST /1/payments`
  - Request body:
    - the body will be a JSON representation of a payment, *without* the `id`. This field will be generated by the API
  - Responses:
    - `201` if the payment was successfully created, the response body will formatted as JSON. The field `payment_id` will contain the generated UUIDv4 `json { "payment_id": "uuidv4" }`
    - `400` if the JSON is not a valid, see and see
    - `500` for a server error, see
- Update a payment: `PUT /1/payments/${uuidv4}`
  - Request parameters:
    - `uuidv1`: will represent the `id` of a payment
  - Request Body:
    - the body will be a JSON representation of a payment, *without* the `id`. This field will be generated by the API
  - Responses:
    - `204` if the payment was successfully updated
    - `400` if `uuidv4` is not a valid UUIDv2 or if the JSON is not valid, see and see
    - `404` if the payment id `uuidv4` doesn't exist in the storage
    - `500` for a server error, see
- Delete a payment: `DELETE /1/payments/${uuidv4}`
  - Request parameters:
    - `uuidv1`: will represent the `id` of a payment
  - Request Body:
    - the body will be a JSON representation of a payment, *without* the `id`. This field will be generated by the API
  - Responses:

        - `204` if the payment was successfully deleted
        - `400` if `uuidv4` is not a valid UUIDv4, see
        - `404` if the payment id `uuidv4` doesn't exist in the storage
        - `500` for a server error, see
- List all payments: `GET /1/payments`
    - Responses:
        - `200`: the response body will be formatted as JSON. The `payments` attribute will be an array of payment, as one from the examples: `json { "payments": [...] }`
        - `500` for a server error, see
- Check if the server is alive: `GET /isalive`
    - Responses:
        - `200`: the response body will be formatted as JSON. The `payments` attribute will be an array of payment, as one from the examples: `json { "yes": "i_am" }`
        - `500` for a server error, see

**Common HTTP answers**

All `4XX` and `5XX` errors will have the same JSON response body `json { "code": "$code", "message": "$message" }`

- `$code` will represent the http error code as an integer. Some developers prefer to also have the error code in the response body
- `$message` will be a string representing the error. Either a error validation message or a `Internal server error` for a `500`

# Further evolution

As we are dealing with payments, in a real world case we would need to implement authentication and rights managements. Depending on the needs, I would either go for: * Basic HTTP auth * Signature of the incoming calls, with a shared secret * Use of a well known auth mechanism (oauth2, or any other)

If we want to continue to use HTTP, of course all requests must at least be encrypted by TLS so HTTPS.

I would also have a stronger validation of the incoming data. It seems a lot of data stored as `string` are something else (enumerations, big decimal, dates, addresses, etc.)

We could provide an openapi representation of the API. I will both provide documentation internally, and also be sharable to customers for them to implement the API.

For the storage, I think the filesystem is a good trade off, it provides atomicity, but it doesnt scale well as it's only on one machine. If the API would require more scalability I would go for a RDBMS. As we are dealing with payments the ACID properties ensured my a standard SQL database is a must have.

For the API to be production ready, I would at least add: * better logging * metrics reporting (latency to storage, latency of the full API, number of requests, etc.)

# Annexe

## JSON schema of the payments

Based on the provided examples of payments, I inferred the schema of a payment to look like this:

### Payment

- type: `string`
- id: `string`, representing a UUIDv4
- version: `unsigned integer`
- organisation_id: `string`, representing a UUIDv4
- attributes: `object`, see

### Payment attributes

- amount: `string`, I inferred it's a string representation of a big decimal, but I'll use `string` for the sake of this test
- beneficiary_party: `object`, see
- charges_information: `object`, see
- currency: `string`
- debtor_party: `object`, see
- end_to_end_reference: `string`
- fx: `object`, see
- numeric_reference: `string`
- payment_id: `string`
- payment_purpose: `string`
- payment_scheme: `string`
- payment_type: `string`
- processing_date: `string`, representing a date in the format `YYYY-MM-DD`
- reference: `string`
- scheme_payment_sub_type: `string`
- scheme_payment_type: `string`
- sponsor_party: `object`, see

### Payment party

- account_name: `string`
- account_number: `string`
- account_number_code: `string`
- account_type: `integer`
- address: `string`
- bank_id: `string`
- bank_id_code: `string`
- name: `string`

### Charges information

- bearer_code: `string`
- sender_charges: `array`, of charges
- receiver_charges_amount: `string`
- receiver_charges_currency: `string`

**FX**

- contract_reference: `string`
- exchange_rate: `string`, probably also a big decimal
- original_amount: `string`, same as above
- original_currency: `string`

**Charges**

- amount: `string`
- currency: `string`