# Payments API Design

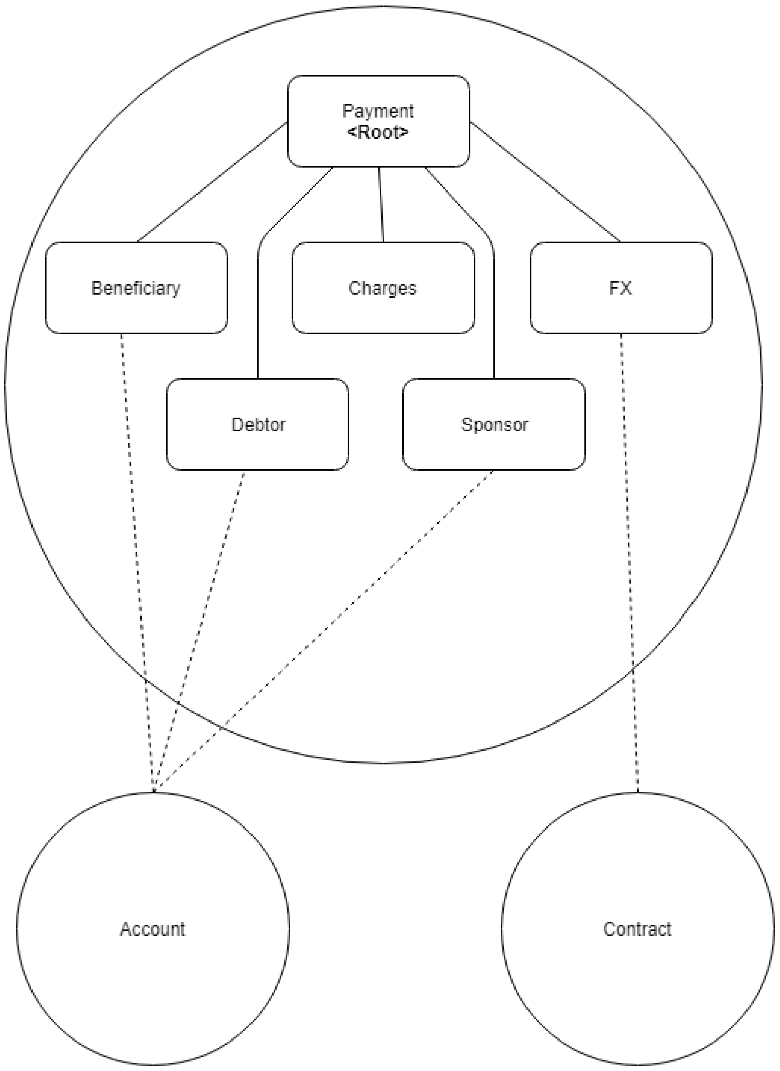
## Domain Definition

A payment might be represented by the following Json:

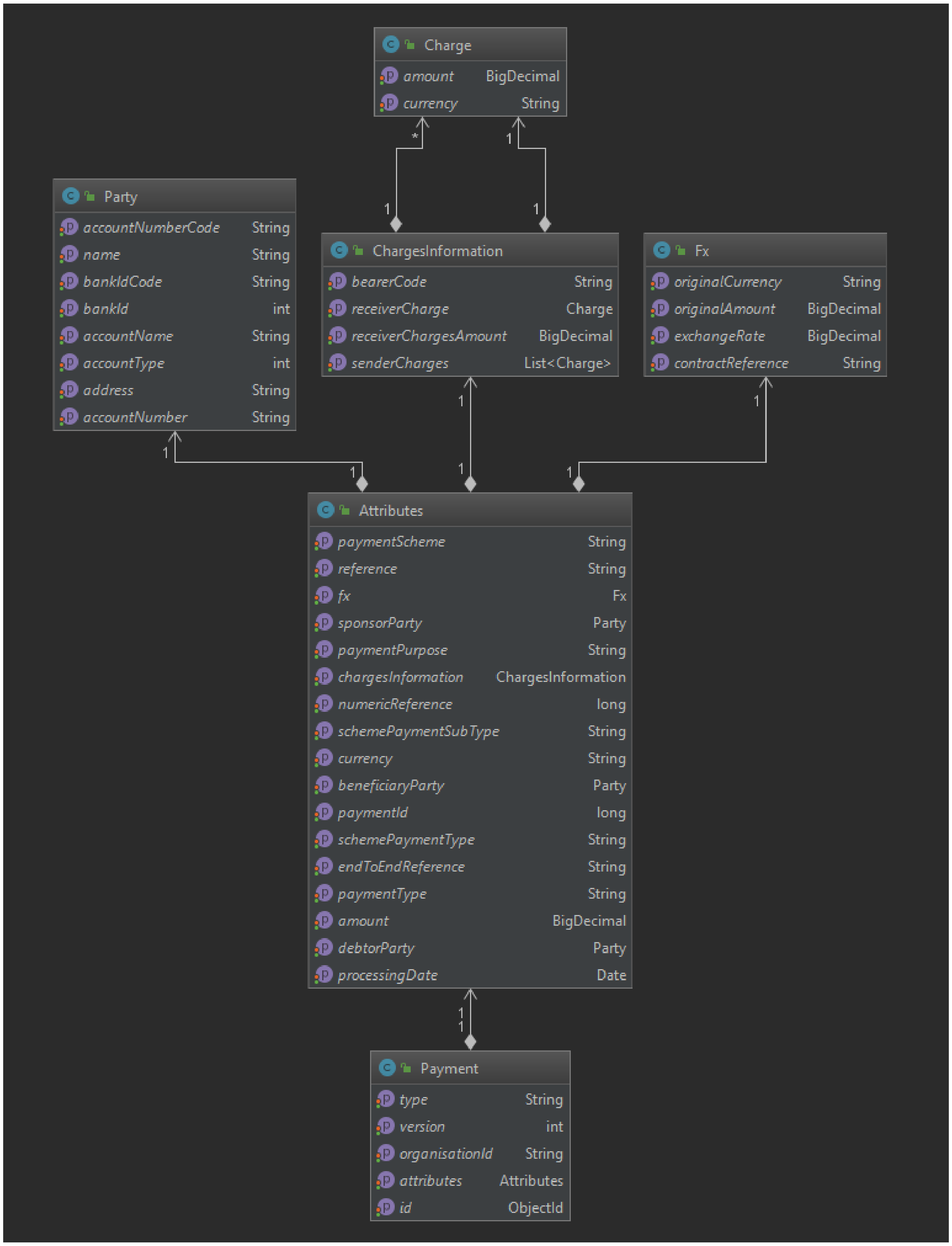
```json
{
  "type": "Payment",
  "id": "4ee3a8d8-ca7b-4290-a52c-dd5b6165ec43",
  "version": 0,
  "organisation_id": "743d5b63-8e6f-432e-a8fa-c5d8d2ee5fcb",
  "attributes": {
    "amount": "100.21",
    "beneficiary_party": {
      "account_name": "W Owens",
      "account_number": "31926819",
      "account_number_code": "BBAN",
      "account_type": 0,
      "address": "1 The Beneficiary Localtown SE2",
      "bank_id": "403000",
      "bank_id_code": "GBDSC",
      "name": "Wilfred Jeremiah Owens"
    },
    "charges_information": {
      "bearer_code": "SHAR",
      "sender_charges": [
        {
          "amount": "5.00",
          "currency": "GBP"
        },
        {
          "amount": "10.00",
          "currency": "USD"
        }
      ],
      "receiver_charges_amount": "1.00",
      "receiver_charges_currency": "USD"
    },
    "currency": "GBP",
    "debtor_party": {
      "account_name": "EJ Brown Black",
      "account_number": "GB29XABC10161234567801",
      "account_number_code": "IBAN",
      "address": "10 Debtor Crescent Sourcetown NE1",
      "bank_id": "203301",
      "bank_id_code": "GBDSC",
      "name": "Emelia Jane Brown"
    },
    "end_to_end_reference": "Wil piano Jan",
    "fx": {
      "contract_reference": "FX123",
      "exchange_rate": "2.00000",
      "original_amount": "200.42",
      "original_currency": "USD"
    },
    "numeric_reference": "1002001",
    "payment_id": "123456789012345678",
    "payment_purpose": "Paying for goods/services",
    "payment_scheme": "FPS",
    "payment_type": "Credit",
    "processing_date": "2017-01-18",
    "reference": "Payment for Em's piano lessons",
    "scheme_payment_sub_type": "InternetBanking",
    "scheme_payment_type": "ImmediatePayment",
    "sponsor_party": {
      "account_number": "56781234",
      "bank_id": "123123",
      "bank_id_code": "GBDSC"
    }
  }
}
```

From the above data definition, here’s a simple schema of the Payment Aggregate Root. We can notice that the Payment aggregate has several relationships to potential other domains:

- Beneficiary, Debtor and Sponsor are related to the Account domain
- Fx is related to the Contract domain



By coupling both Domain Definition and the Json data structure, we can deduce the following class diagram:



## API Design

The Payments API will allow to:

- Fetch a payment resource
- Create, update and delete a payment resource
- List a collection of payment resources
- Persist resource state (in database)

**Conventions**:

- The API will be versioned and thus has the following base path: ***/api/v1***
- The resource name will be ***payments***
- A given payment will be identified by its ***id***

As a summary, the microservice will follow this design:

| Resource | POST | GET | PUT | DELETE |
|---|---|---|---|---|
| /api/v1/payments | Create a payment | List payments | **/** | **/** |
| /api/v1/payments/:id | **/** | Fetch a payment | Update a payment | Delete a payment |

Below is the OpenApi V3 definition for all the methods listed above.

### Fetch a payment resource

```
/payments/{id}:
  get:
    tags:
      - "payments"
    summary: "Get payment by ID"
    description: "Returns a single payment"
    operationId: "getPayment"
    produces:
      - "application/json"
    parameters:
      - name: "id"
        in: "path"
        description: "ID of payment to return"
        required: true
        type: "string"
    responses:
      200:
        description: "Successful operation"
        schema:
          $ref: "#/definitions/Payment"
      404:
        description: "Not found"
```

### Create a payment resource

```
/payments:
  post:
    tags:
      - "payments"
    summary: "Create a new payment"
    description: ""
    operationId: "addPayment"
    consumes:
      - "application/json"
    produces:
      - "application/json"
    parameters:
      - in: "body"
        name: "body"
        description: "Payment object"
        required: true
        schema:
          $ref: "#/definitions/Payment"
    responses:
      201:
        description: "Created"
      400:
        description: "Bad request"
      415:
        description: "Unsupported media type"
```

## Update a payment resource

Even tough it becomes more and more usual to find API's that are either updating or creating a resource if this one does not exist yet, we'll stick here to the traditional definition of a PUT:

- Try to find the resource by ID
- If it exists, update it and return a 200 code
- If it doesn't, throw a 404

```
/payments/{id}:
    put:
    tags:
      - "payments"
    summary: "Update an existing payment"
    description: ""
    operationId: "updatePayment"
    consumes:
      - "application/json"
      - "application/xml"
    produces:
      - "application/xml"
      - "application/json"
    parameters:
      - name: "id"
        in: "path"
        description: "ID of payment to return"
        required: true
        type: "string"
      - in: "body"
        name: "body"
        description: "Payment object that needs to be updated"
        required: true
        schema:
          $ref: "#/definitions/Payment"
    responses:
      200:
        description: "Successful operation"
      404:
        description: "Not found"
```

```yaml
      415:
        description: "Unsupported media type"
```

### Delete a payment resource

```yaml
/payments/{id}:
  delete:
    tags:
      - "payments"
    summary: "Deletes a payment"
    description: ""
    operationId: "deletePayment"
    produces:
      - "application/json"
    parameters:
      - name: "api_key"
        in: "header"
        required: false
        type: "string"
      - name: "id"
        in: "path"
        description: "Payment id to delete"
        required: true
        type: "string"
    responses:
      204:
        description: "No content"
      404:
        description: "Not found"
```

### List a collection of payment resource

```yaml
/payments:
  get:
    tags:
      - "payments"
    summary: "List all payments"
    description: "Returns a list of all payments"
    operationId: "getAllPayments"
    produces:
      - "application/json"
    responses:
      200:
        description: "Successful operation"
        schema:
          $ref: "#/definitions/ArrayOfPayments"
```

## Technical choices

In order to speed up the development process, the following framework stack will be considered:

1. Spring boot for a quick setup
2. Spring Data Rest for the Data modelisation and Jackson mapping
3. Spring HATEOS for API discoverability
4. Spring Validation in order to validate API inputs (@Valid annotation)
5. Swagger in order to document the API
6. Lombok to reduce boiler plate Pojo’s code
7. Spring Test for both TDD and BDD

8. Spring actuator in order to provide basic monitoring information

Concerning the database persistence, one straightforward consideration might be using Spring boot data JPA with an embedded H2 database for example.

However, since I’ve never used it before, I’ll give a try to Spring boot data MongoDB and an embedded Mongo database.

Our API should be discoverable in order to facilitate front-end developments: let’s use Spring HATEOAS in order to add resources links.

## Possible improvements

- Currently, the model is validated based on my own perception: some fields have been put as mandatory, currencies should be defined in ISO code, etc.
- This would be possible to use a CQRS / event sourcing in order to decouple writing and reading (and to capitalize on using the same structure for other microservices). Currently considered out of scope for the current exercise.