API Design

For the purpose of this exercise I am going to simplify the payment data model to something as simple as:

```
payment {
        id      : string
        from    : string
        to      : string
        value   : integer
}
```

The actual API will evolve around the HTTP request methods and end-points:

```
GET /payments               > return all payments
GET /payments/{id}          > return payment with id
DELETE /payments/{id}       > delete payment with id
POST /payments              > insert a new payment (id assigned on the server side)
PUT /payments/{id}          > update payment with id
```

Also, all objects are represented using JSON format

Now there are multiple aspects which can be discussed here:
- Api versioning, for that we can prefix it (e.g. api/v1…)
- PUT vs POST based on idempotence etc.. In this case I decided to go with using POST for creation and PUT for update.
- Complex queries with GET with filters e.g. /payments?from=…
- Maybe use PATCH to update only parts of the payment objects
- For objects containing collections of other objects like payment properties etc.. we can extend the API as follow /payments/{id}/properties/{prop_id} or we can have a parallel api like / properties/{prop_id}

Further concerns could include how do we implement this API, security (authentication & authorisation), persistence of the objects, how do we deploy it into production, how do we test it. Even more advanced concerns could include services discovery, stability and resilience, etc…

For the implementation, I am going to use Spring Boot WebFlux, the new web framework on the reactive stack (as anyway I never used Spring Boot MVC). Persistence is handled using Spring Data and reactive MongoDB driver and MongoDB database (all tests are going to use the embedded MongoDB version, for running the application make sure you start a MongoDB daemon on the same machine & default settings). Spring Boot can help also with testing, especially higher level testing like the end points. Further, security (with both of its aspects) can be handled using Spring Security (not included in the current demo).

One final note is about the choice of the type for the id field. For this, I used String just to simplify the work for the scope of the demo.

Implementation of the this pico service

```
GatewayApplication            – application entry point
entities/Payment              – the domain model (in this case the payment)
repositories/PaymentRepository– handles the persistence of the payments
controllers/PaymentController – handles the actual mapping of the urls to the
                                business logic
```

Tests run against embedded MongoDB.

Trying the application with curl:

Before running the application, make sure you have a MongoDB instance up and running.

```
Create payment:
curl –v –X "POST" "http://localhost:8080/payments" \
       –H "Accept: application/json;charset=UTF–8" \
       –H "Content–Type: application/json;charset=UTF–8" \
       –d $'{"from":"fromC","to":"toC","amount":1}'

Update payment:
curl –v –X "PUT" "http://localhost:8080/payments/some_id" \
       –H "Accept: application/json;charset=UTF–8" \
       –H "Content–Type: application/json;charset=UTF–8" \
       –d $'{"from":"fromU","to":"toC","amount":2}'

Delete payment by id:
curl –v –X "DELETE" "http://localhost:8080/payments/some_id" \
     –H "Accept: application/json;charset=UTF–8”

Fetch payment by id:
curl –v –X "GET" "http://localhost:8080/payments/some_id" \
     –H "Accept: application/json;charset=UTF–8"

Fetch all payments:
curl –v –X "GET" "http://localhost:8080/payments" \
     –H "Accept: application/json;charset=UTF–8"
```