Payment API

Requirements

The Payment API is implemented in `Spring Boot` using REST Controller . It is build by `gradle`. `PostgreSQL` is used for storage. There is no ORM involved (e.g. `Hibernate`) as they will be no help of using it. `Spring Data` is also not used as direct mapping is more efficient, I think.

Design

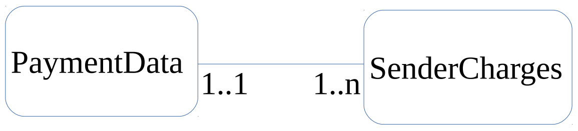
You can see 3 layers on the following Design Diagram. The top most is UI layer of multiple clients (`curl` in our case). Services layer is the back-end serving the CRUD operations to the database. We use feign client from the `Spring Boot Cloud` repository to fetch the external data from http://mockbin.org/bin/41ca3269-d8c4-4063-9fd5-f306814ff03f . It might be considered as too expensive technique, but we can use the Payment API to do it transparently at least. Services layer uses a single `PaymentController` to handle REST requests. The `PaymentController` has the following endpoints:

```
Fetch Payments:   GET    /v1/payments/fetch/payments/fetch/41ca3269-d8c4-4063-9fd5-f306814ff03f
Create Payment:   POST   /v1/payments
Update Payment:  PUT    /v1/payments/<id>
Get Payment:        GET    /v1/payments/<id>
Delete Payment:   DELETE /v1/payments/<id>
```

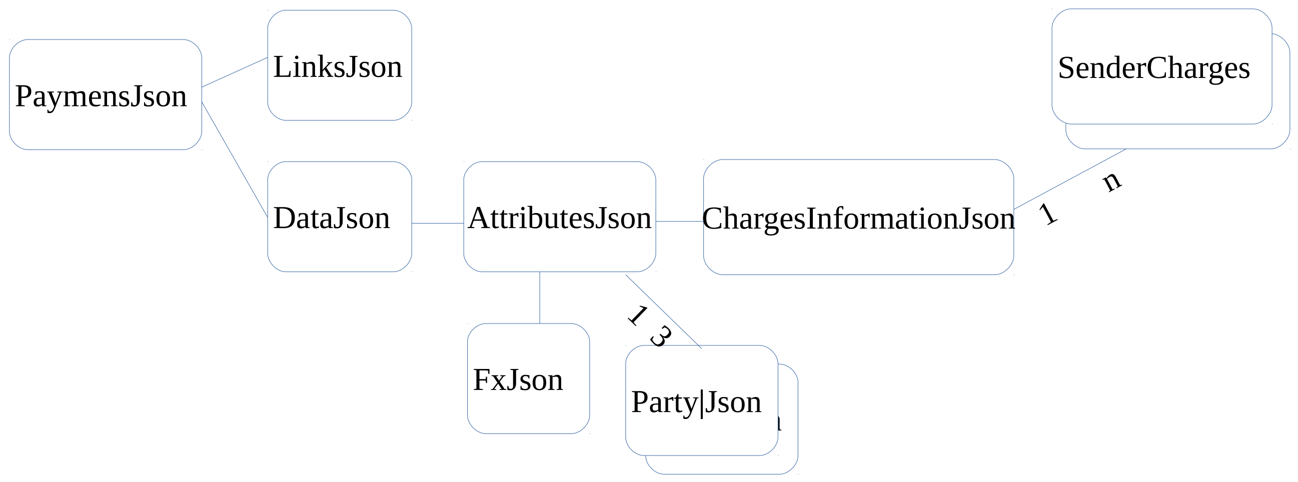
`PaymentService` manages and processes controller’s requests. It contains the most of the business logic. It splits requests into two separate persistent components `PaymentRepository` and `SenderChargesRepository` due to the fact that the input data have 1 to N relationships for `SenderCharges`. The service make use of `PaymentJsonTransformer` and `SenderChargesJsonTransformer` to convert between external and internal models. Note that the internal model is very flat but doesn’t violate any normal form.

Persistence is managed by `PaymentRepository` and `SenderChargesRepository`. The former repository persists data of the main model whereas the latter is supposed to persist only sender charges related to the main model entity with id provided. To do so, the sender charges have a foreign key to the top most entity. Select operations are mapped by `PaymentDataRowMapper` and `SenderChangesRowMapper` for the respective repositories. Note that both repositories don’t support full CRUD as `PaymentRepository`’s update can be replace by transactional DELETE and CREATE and `SenderChargesRepository` doesn’t need all CRUD operations.

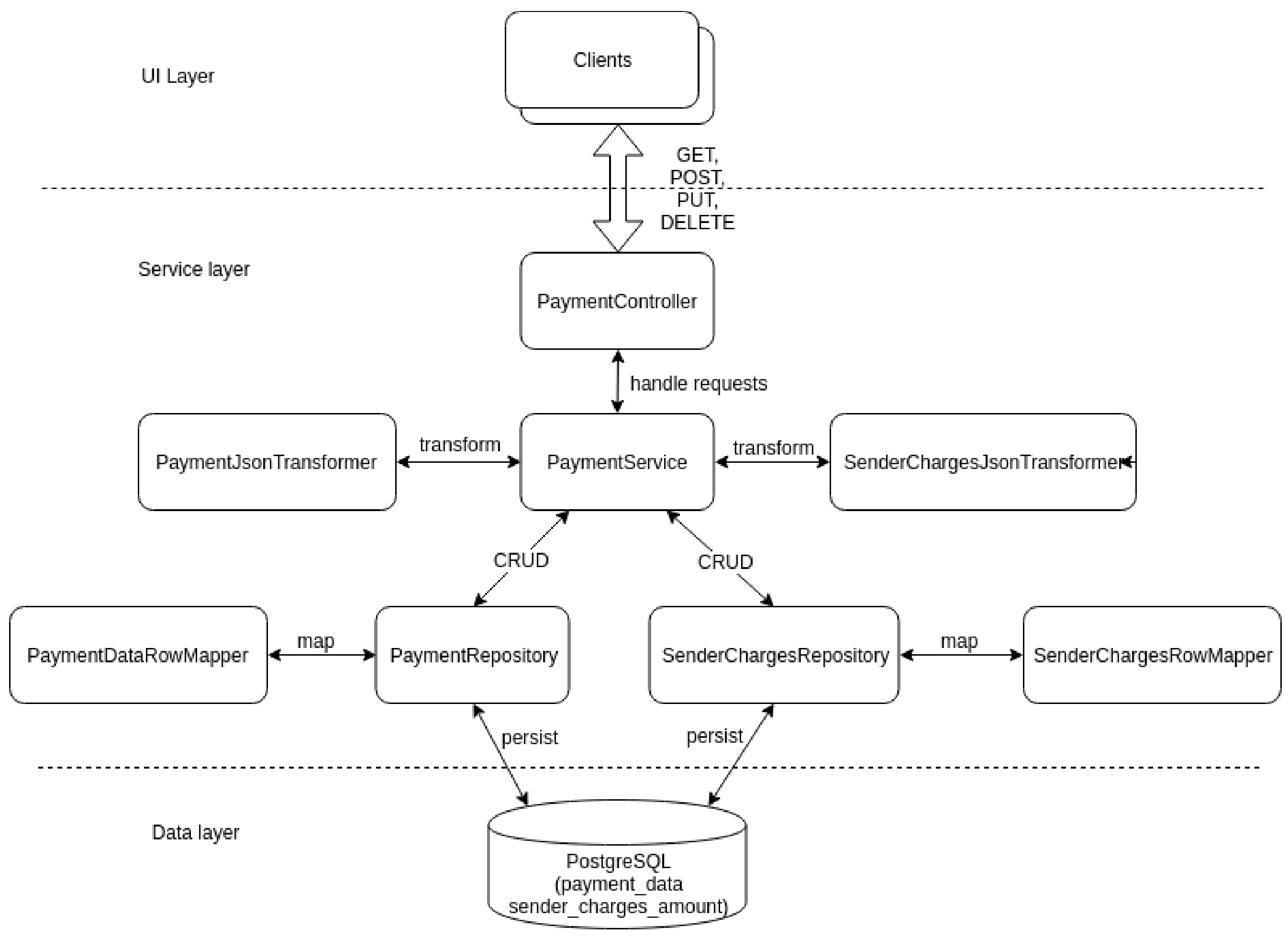
Database Model (Internal)



External Model



Design Diagram


UI Layer
Clients
GET,
POST,
PUT,
DELETE
Service layer
PaymentController
handle requests
PaymentJsonTransformer
transform
PaymentService
transform
SenderChargesJsonTransformer
CRUD
CRUD
PaymentDataRowMapper
map
PaymentRepository
SenderChargesRepository
map
SenderChargesRowMapper
persist
persist
Data layer
PostgreSQL
(payment_data
sender_charges_amount)