Technical Test Backend

My explanation of my solution on your technical backend test.

Introduction

My solution is built using a framework called **Apache Camel** (https://camel.apache.org).
It is based on the Spring Boot template provided, but I decided to use my preferred framework.

This solution everything is based on communication between endpoints; a call to a REST endpoint with a payload that then queries a database and returns data to the caller, or calling other REST services.

I have made the solution light, it could have more error handlers and transaction handling.

Null checks as conditions for further processing is also not *production material*, but this is just a PoC.

For an explaination of the requests, please view the file `rest.http`.

rest.http

```
### Get a Wallet
# @name GetWallet
POST {{host}}/wallet
Content-Type: application/json

{
  "walletId": "516beebc-6c01-4794-af53-08b4793c5ed7"
}

### Topup a wallet
# @name WalletTopup
#
POST {{host}}/wallet/topup
Content-Type: application/json

{
  "walletId": "516beebc-6c01-4794-af53-08b4793c5ed7",
  "amount": 123,
  "cardNumber": "1234567890123"
}

### Show api
# @name api
#
GET {{host}}/api

###
```

Wallet service

This solution is based on a REST interface where a request is sent to an endpoint, this called `/wallet`.

The request is defined and starts in the file `WalletRestRoute.java`.

WalletRestRoute.java

```
/**
 * The class WalletRestRoute
 *
 * @author maw, 2022-04-08
 * @version 1.0
 */
@Component
@CommonsLog
public class WalletRestRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        restConfiguration() ①
                .component("servlet")
                .bindingMode(RestBindingMode.json)
                .dataFormatProperty("prettyPrint", "true")
                .apiContextPath("/api") ②
                .apiProperty("api.title", "Wallet Service")
                .apiProperty("api.description", "API for querying wallet and adding
funds")
                .apiProperty("api.version", "1.0.0")
                .apiProperty("api.contact.email", "mikael.andersson.wigander@pm.me")
                .apiProperty("api.contact.brokerUrl", "https://hmpg.net/")
                .apiProperty("api.license.name", "Copyright Wally Services")
                .apiProperty("cors", "true");

        rest("/wallet") ③
                .consumes("application/json")
                .produces("application/json")
                .description("REST-WALLET", "This is a Wallet service", "en")

                .post() ④
                .type(WalletRequest.class)
                .outType(Wallet.class)
                .to("direct:wallet")

                .post("/topup") ⑤
                .id("WALLET-TOPUP")
                .description("REST-WALLET-TOPUP", "Adding funds to the wallet using a
credit card", "en")
                .type(WalletTopupRequest.class)
                .outType(Wallet.class)
                .to("direct:wallet-topup");
    }
}
```

① This is the configuration of the routing information for the REST.

② Here's the Swagger api documentation part.

③ The root of the context

④ **Wallet service** is using a `POST` all over to disclose any information in URL's. It accepts a structure based on the `WalletRequest.class` and outputs data in the form of `Wallet.class`, but as json. The call continues the to `("direct:wallet")` in class `WalletRoutes.java`

⑤ This is the *Topup* endpoint, built in the same manner.

So the request is processed as a `POST` and then transferred to the `direct` endpoint. In the request is a payload of a `json` structure which is sent along to the next endpoint.

WalletRoutes.java

```
        from(direct("wallet"))
                .routeId("WALLET-ROUTE")
                .description("This is a wallet route")
                .process(exchange -> { ①
                    final Message in = exchange.getIn();
                    final WalletRequest walletRequest = in.getBody(WalletRequest.
class);

                    final Optional<String> walletIdOpt = Optional.ofNullable
(walletRequest.getWalletId());
                    walletIdOpt.ifPresentOrElse(s -> in.setHeader("walletId", s), () -
> in.setBody(null));
                })
                .choice()
                .when(body().isNotNull())
                .setBody(simple("select * from wallet where wallet_id = :?walletId"))
②
                .toD(jdbc("default").outputClass(Wallet.class.getName()) ③
                                    .outputType(JdbcOutputType.SelectOne)
                                    .useHeadersAsParameters(true))
                .log("${body}")
                .end();
```

① The payload is processed as a `WalletRequest` and the wallet ID is set as a header.

② The payload/body is set to the SQL we'd like to be executed.

③ The `JDBC` component executes the SQL and returns zero or one record and if a record is found it will be of type `Wallet.class`.

Wallet.java

```
/**
 * The class Wallet
 *
 * @author maw, 2022-04-09
 * @version 1.0
 */
@Entity(name = "wallet")
@Table(name = "wallet")
@Data
```

```java
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Wallet implements Serializable {
    /**
     * The Id.
     */
    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @JsonProperty
    long id;

    /**
     * The Wallet id.
     */
    @Column(name = "wallet_id")
    @JsonProperty
    String walletId;

    /**
     * The Balance.
     */
    @Column
    @JsonProperty
    BigDecimal balance;

    /**
     * The Created.
     */
    @Transient
    @JsonProperty
    Timestamp created;

    /**
     * The Updated.
     */
    @Transient
    @JsonProperty
    Timestamp updated;
}
```

Wallet Topup service

This service is a little more complex and not that straight forward as the Wallet service.

This solution is based on a REST interface where a request is sent to an endpoint, this called `/wallet/topup`.

Here we have more steps to control and process.

1. We have a call from a REST endpoint with a payload of a Wallet ID, an amount and a credit card number.
2. We need to retrieve the Wallet for the provided ID.
3. We need to use the Stripe service to charge the card given amount.
4. Update the wallet with the new amount.

WalletRoutes.java

```
        from(direct("wallet-topup"))
                .routeId("WALLET_TOPUP")
                .description("Topup a wallet with funds")
                .process(exchange -> { ①
                    final Message in = exchange.getIn();
                    final WalletTopupRequest walletTopupRequest = in.getBody
(WalletTopupRequest.class);
                    final Optional<String> walletIdOpt =
                            Optional.ofNullable(walletTopupRequest.getWalletId());
                    walletIdOpt.ifPresentOrElse(s -> {
                        in.setHeader("walletId", s);
                        in.setBody(new WalletRequest(s), WalletRequest.class);
                    }, () -> in.setBody(null));

                    final Optional<BigDecimal> amountOpt = Optional.ofNullable
(walletTopupRequest.getAmount());
                    amountOpt.ifPresentOrElse(s -> {
                        in.setHeader("topUpAmount", s);
                    }, () -> in.setBody(null));

                    final Optional<String> cardNumberOpt =
                            Optional.ofNullable(walletTopupRequest.getCardNumber());
                    cardNumberOpt.ifPresentOrElse(s -> in.setHeader("cardNumber", s),
() -> in.setBody(null));
                })
                .choice()
                .when(body().isNotNull())
                .to(direct("wallet")) ②
                .end()
                .choice()
                .when(body().isNotNull())
                .setHeader("wallet", simple("${body}")) ③
                .bean(StripeService.class, "charge(${header.cardNumber},
${header.topUpAmount})") ④
                .end()
                .choice()
                .when(body().isNotNull())
                .log("${body}")
                .process(exchange -> { ⑤
                    final Message in = exchange.getIn();
                    final Payment payment = in.getBody(Payment.class);
                    final Wallet wallet = in.getHeader("wallet", Wallet.class);
                    wallet.setBalance(wallet.getBalance()
                                              .add(payment.getAmount()));
                    in.setBody(wallet, Wallet.class);
                })
                .toD(jpa(Wallet.class.getName()).useExecuteUpdate(true)) ⑥
                .log("${body}")
                .end();
```

① Incoming payload is processed to retrieve the three parts.

② Since we already have a route for getting the Wallet we use it.

③ We store the Wallet entity as a header for further use later.

④ The Stripe service is called as is, using a bean with arguments. This is to show that we can use *normal* POJO and services as well.

⑤ We process the returning `Payment` and setting the new amount on the `Wallet` entity.

⑥ We then sends this to the `JPA` component with an update request and the data is persisted.

WalletTopupRequest.java

```
/**
 * The class WalletTopupRequest
 *
 * @author maw, 2022-04-10
 * @version 1.0
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class WalletTopupRequest {
    /**
     * The Card number.
     */
    private String cardNumber;
    /**
     * The Amount.
     */
    private BigDecimal amount;
    /**
     * The Wallet id.
     */
    private String walletId;
}
```