Übung 5: GraphQL Bank

In dieser Übung soll die *Bank* aus den vorangehenden Übungen mit Hilfe von GraphQL realisiert werden. Dazu muss als erstes ein GraphQL-Schema definiert werden, welches die Datentypen und die Query- und die Mutations-Operationen definiert. Die Informationen eines Kontos können z.B. mit folgendem Datentyp repräsentiert werden:

```
type Account {
   number: ID!
   owner: String!
   active: Boolean!
   balance: Float!
}
```

Für die Implementierung des Servers kann das Projekt aus dem Unterricht (*08_GraphQL*) als Ausgangspunkt verwendet werden.

Aus den Methoden der Driver-Implementierung müssen dann auf die URL http://localhost:8080/graphql POST-Requests abgesetzt werden. Dazu wird am besten die Klasse `HttpClient` aus dem JDK verwendet.

Ihre Server-Implementierung können Sie mit der GraphiQL-Schnittstelle testen, z.B. mit folgendem Query:

```
query {
   getAccount(number: "101-47-000") {
      owner
      balance
      active
  }
}
```

Alternativ könnten Sie auch mit den Command-Objekten arbeiten, dann vereinfacht sich das Schema zu

```
type Query {
   hello: String    # ein Query ist zwingend vorgeschrieben
}

type Mutation {
   execute(command: String!): String!
}

schema {
   query: Query
   mutation: Mutation
}
```

wobei bei dieser Definition das Kommando als String übertragen werden muss (GraphQL kennt keine Byte-Arrays). Ein Byte-Array können Sie mit dem Base64-Encoding-Schema in einen String umwandeln. Mit diesem Ansatz können Sie jedoch keine interaktiven Abfragen über die GraphiQL-Schnittstelle machen.

Grundsätzlich würde GraphQL auch Subscriptions (über WebSockets) ermöglichen, und damit könnte das `BankDriver2`-Interface (mit Notifikationen) implementiert werden. Dazu muss mit dem Property

```
spring.graphql.websocket.path=/graphql
```

der Websocket-Endpunkt definiert und freigeschaltet werden, und dann muss im Klienten eine Websocket-verbindung auf `ws://<host>:<port>/graphql` aufgebaut werden (mit dem WS-Subprotokoll `graphql-transport-ws`). Sobald die Verbindung steht, müssen folgende Text-Meldungen an den Server geschickt werden:

```
{
    "type"    : "connection_init",
    "payload" : {}
}
```

gefolgt von der eigentlichen Subscription in der Form

```
{
    "id"      : "2ea12068-7139-4f77-9e13-298642b66d7e",
    "type"    : " subscribe",
    "payload" : {"query":"subscription{…}"}
}
```

wobei der Parameter `id` eine beliebige Identifikation des Klienten ist.

Auf dem Server müssen zwei Variablen vom Typ `Sinks.Many<T>` und `Flux<T>` deklariert werden. Wenn Sie als Resultat auf eine Subscription nicht nur die Kontonummer, sondern gerade das geänderte Konto zurückgeben, dann heisst das, dass die beiden Variablen

```
Sinks.Many<Account> sink;
Flux<Account> flux;
```

deklariert und mit

```
sink = Sinks.many().multicast().onBackpressureBuffer(
       reactor.util.concurrent.Queues.SMALL_BUFFER_SIZE,
       false
);
flux = sink.asFlux();
```

initialisiert werden müssen. Auf dem Sink können bei Änderungen mit der Methode `sink.tryEmitNext(…)` Änderungen publiziert werden, und innerhalb der Subscription-Methode kann dann der Flux zurückgegeben werden:

```
@SubscriptionMapping
public Publisher<Account> subscribeChangeNotification() {
       return flux;
}
```