Übung 3: REST Bank

In dieser Übung soll die Bank aus den vorherigen Übungen mit REST realisiert werden. Dabei soll eine REST-konforme Architektur (ROA – Resource Oriented Architecture) realisiert werden, d.h. die Bank-Operationen sollen auf GET/PUT/POST/DELETE- und PATCH-Methoden auf Ressourcen abgebildet werden.

Auf der URL http://host:port/bank/accounts sollen z.B. folgende HTTP-Methoden angeboten werden:

GET	Abfrage der Konten (Liste von URLs)
POST	Erzeugen eines neuen Kontos, die neue URL (Kontonummer) kann im Location-Header-Feld einer `203 Created`-Antwort zurückgegeben werden

Auf der URL http://host:port/bank/accounts/{id} könnte man sich folgende Operationen vorstellen:

GET	Abfrage der Kontodaten für ein bestimmtes Konto
PUT/PATCH	Aktualisierung des Saldos
DELETE	Löschen des Kontos
HEAD	Abfrage, ob Konto noch aktiv ist, falls Antwort `410 Gone`, dann ist das Konto geschlossen worden, falls die Antwort `404 NotFound` lautet, dann ist die Kontonummer ungültig. Die Methode GET müsste dann dieselben Header zurückgeben, d.h. auf einem geschlossenen Konto könnte dann weder der Saldo (dieser ist in diesem Fall jedoch 0) noch der Name des Kontoinhabers abgefragt werden.

Interessant ist die Frage, wie man die Transfer-Methode in einer Ressourcen-orientierten Architektur (ROA) implementiert, denn die Transfer-Funktion operiert auf zwei unterschiedlichen Ressourcen. Beachten Sie dabei, dass die Methoden GET, PUT und DELETE idempotent sein müssen, d.h. mehrfache Hintereinanderausführung einer Methode mit denselben Argumenten muss auf dem Server denselben Effekt haben. Eine einfache Lösung ist, das Transfer mit einem POST/PATCH-Request zu implementieren. Sehen Sie auch eine Möglichkeit diese Operation idempotent zu realisieren? (=> Optimistic Locking)

Um die Datenmenge, die bei einem GET-Request auf bank/accounts/ übertragen wird, gering zu halten, können Sie ein Conditional-GET einsetzen.

Als Umgebung für die Entwicklung des Servers empfehle ich Ihnen Jersey, die Referenzimplementierung von *Jakarta RESTful WebServices* (https://eclipse-ee4j.github.io/jersey/). Sie finden auf der Jersey Webseite Dokumentationen und Beispiele. *Jakarta RESTful WebServices* bietet auch Unterstützung für die Implementierung von Klienten an, d.h. es kann auch auf Klientenseite verwendet werden, um auf REST-Ressourcen zuzugreifen (diese Funktionalität wird in den Proxy-Klassen benötigt). Alternativ kann auch Dropwizzard (https://www.dropwizard.io/) verwendet werden welches mehrere Bibliotheken zu einem schlanken Framework kombiniert.

Alternativ könnte auch Javalin (https://javalin.io) verwendet werden, welches allerdings nicht kompatibel zur Spezifikation *Jakarta RESTful WebServices* ist. Eine weitere Alternative wäre die Verwendung von Spring Boot (https://spring.io/projects/spring-boot), aber auch bei Spring Boot müssen die REST-Schnittstellen anders definiert werden.

Auf Klientenseite müssen wiederum Proxy-Klassen implementiert werden. Dabei müssen HTTP GET-, PUT-, POST-, DELETE-, HEAD- und allenfalls PATCH-Methoden abgesetzt werden. Dazu stehen folgende Varianten zur Verfügung:

- Jersey Client Implementation (auf der Basis von *Jakarta RESTful WebServices*)
- Java HttpClient den Sie bereits in Übung 2 verwendet haben

Sie finden im Netz bei Bedarf weitere REST-Client Implementierungen.

Besprechung: 31. März / 1. April 2025