Übung 4: WebSockets Bank

In dieser Übung soll die *Bank* aus den vorhergehenden Übungen mit Hilfe von WebSockets implementiert werden, inzwischen eigentlich keine spannende Aufgabe mehr, daher wollen wir Stufe 2 unserer Beispielanwendung zünden!

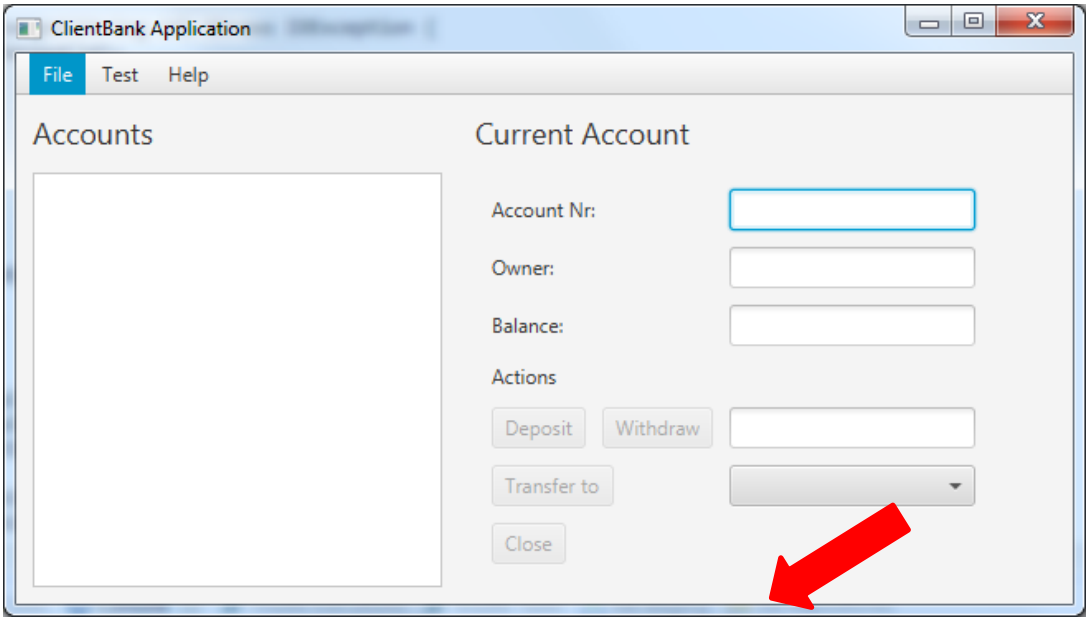
Bisher haben Sie in Ihren Übungen auf Klientenseite jeweils das Interface `BankDriver` implementiert. Neben diesem Interface gibt es im Projekt, das wir Ihnen abgegeben haben, auch noch das Interface `BankDriver2` (als Erweiterung des Interfaces `BankDriver`). Dieser Driver erlaubt es, dass ein *Listener* registriert wird (Interface `UpdateHandler`), welcher bei Änderungen aufgerufen wird. Als Parameter der Methode `accountChanged` wird die Kontonummer des neuen, geänderten oder gelöschten Kontos mitgegeben. Nach einem erfolgreichen Aufruf der `transfer`-Methode gibt es zwei Notifikationen (da sich zwei Konten geändert haben).

```
public interface BankDriver2 extends BankDriver {

	void registerUpdateHandler(UpdateHandler handler) throws IOException;

	interface UpdateHandler {
		void accountChanged(String id) throws IOException;
	}
}
```

Wenn Sie dieses Interface in Ihrer Driver-Implementierung implementieren, dann gibt es im GUI *keinen* Refresh-Knopf mehr, sondern die Daten werden im GUI bei jeder Änderung *automatisch* aktualisiert. Dazu registriert die Klasse `bank.gui.Client` einen Update-Handler, der bei jeder Änderung eines Kontos das GUI aktualisiert.



WebSockets sind ideal, um asynchron Informationen vom Server an die aktuell verbundenen Klienten zu senden – in unserem Fall Informationen über die Änderung von Kontodaten. Aus diesem Grund soll in dieser Übung das Interface `bank.BankDriver2` implementiert werden.

Als WebSocket-Implementierung können Sie die Referenzimplementierung Tyrus [1, 2] verwenden, welche sowohl den Klienten als auch den Server unterstützt. Auf Klientenseite können Sie alternativ auch den im JDK vorhandenen HTTP-Klienten verwenden, und auf Serverseite könnten Sie alternativ auch Tomcat verwenden.

Bei der Implementierung der synchronen Bank-Methoden (wie `createAccount` oder `deposit`) ist zu beachten, dass die Antwort auf eine WebSockets-Anfrage *asynchron* zurückgegeben wird. Der aufrufende Thread muss also warten, bis die Antwort in der `onMessage`-Methode zurückgegeben wird, wobei diese Methode von einem anderen Thread ausgeführt wird. Man muss daher Daten zwischen zwei Threads austauschen können. Beachten Sie auch, dass die Antwort auch eine Änderungsnotifikation sein kann, die nichts mit der erwarteten Antwort zu tun hat!

Für die Notifikation aller Klienten könnte die in der Klasse `Session` definierte Methode

```
Set<Session> getOpenSessions()
```

nützlich sein. Diese Methode liefert eine Menge aller offenen Verbindungen.

Falls Sie mit Command-Objekten arbeiten, die zwischen Client und Server ausgetauscht werden, dann könnte die Konvertierung einer binären Meldung und einem Request-Objekt in einen Encoder/Decoder ausgelagert werden. Die Interfaces, die sie dazu implementieren müssen, sind

```
javax.websocket.Encoder.Binary<T>
javax.websocket.Decoder.Binary<T>
```

Die Implementierung auf der Serverseite sieht dann (für `T = Command`) wie folgt aus:

```
@OnMessage
public Command onMessage(Command cmd) {
   return cmd.execute(bank);
}
```

[1] https://projects.eclipse.org/projects/ee4j.tyrus
[2] https://eclipse-ee4j.github.io/tyrus-project.github.io/documentation/2.2.0/index/