

# Verteilte Systeme

## Übung E

Bearbeitungszeit 2 Wochen

Diese Übung adressiert den Umgang mit JAX-B und JAX-WS, sowie den Umgang mit lokalen Transaktionen in relationalen Datenbanken mittels JDBC. Siehe auch

- [http://en.wikipedia.org/wiki/Java\\_API\\_for\\_XML\\_Web\\_Services](http://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services)
- <http://en.wikipedia.org/wiki/JAX-B>
- <http://de.wikipedia.org/wiki/JDBC>

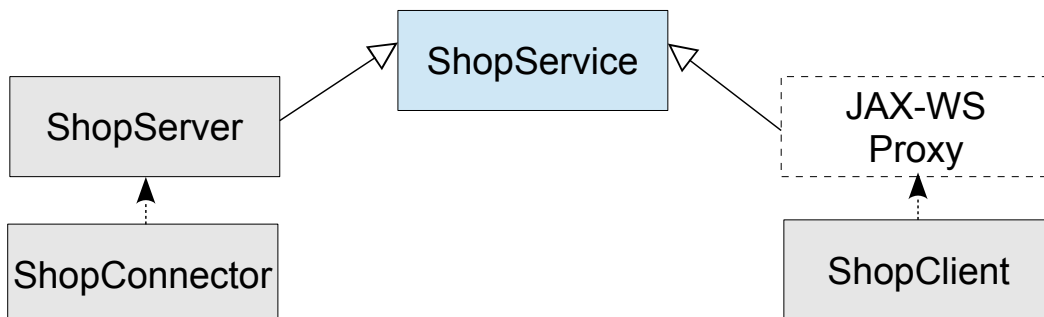
### Aufgabe 1: Aufsetzen der Shop-Datenbank

Verwendet den Inhalt des Scripts `shop-mysql.ddl` um das Datenbankschema `soap_shop` in MySQL zu definieren und zu initialisieren; am einfachsten per Copy&Paste nachdem ihr Euch im Kommando-Interpreter mittels

```
mysql --host=<host> --port=<port> --user=<userid> --password=<password>
```

angemeldet habt; Falls der Host dabei localhost und/oder der Port 3306 ist, könnt ihr die entsprechenden Argumente weglassen. Kopiert danach die Datei `mysql-connector-java-5.1.x-bin.jar` aus dem lib- Verzeichnis des Verteilte Systeme-Ordners auf Euer System, und erweitert den Class-Path Eures Java-Projektes um diese Datei. Die Datei `shop-mysql.properties` enthält zudem die Datenbank-Zugangsdaten für den Datenbank-Zugriff aus Java, diese müssen je nach Datenbank-Setup in situ angepasst werden.

### Aufgabe 2: Service Interface und Interface-Halo



Benennt alle ...Skeleton Klassen so um dass „Skeleton“ aus dem Namen verschwindet.

Fügt im Service-Interface *ShopService* alle JAX-WS Annotationen hinzu die für einen bottom-up generierten Web-Service notwendig sind. Dazu soll:

- das Service-Interface mit `@WebService` als solches annotiert werden
- jeder Parameter mit `@WebParam(name="<name>")` annotiert werden
- die Methode `cancelOrder(...)` zudem mittels `@Oneway` als asynchron annotiert werden

**Beachtet:** Statt `@WebParam` könnte auch `@XmlElement` annotiert werden um die Parameternamen zu setzen. Diese Annotation bietet sogar weitergehende Möglichkeiten um Kardinalität und Nullbarkeit von Parametern/Resultaten in der WSDL zu definieren; leider jedoch werden diese Einstellungen von den generierten Endpoints und Client-Proxies ignoriert, was folgende Auswirkungen hat:

- Der Mechanismus null-Parameter/Resultate in SOAP zu übertragen ist stets das entsprechende XML-Element bei der Übertragung entfallen zu lassen, egal was die WSDL an Constraints definiert.
- Parameter und Resultate die nicht-primitive Einzeltypen aufweisen können daher senderseitig immer als null übergeben werden, und erscheinen auf der Empfängerseite dann auch als null. Dies gilt als Erweiterung dieser Regel zudem für `byte[]` und `char[]`, weil diese Typen in WSDL als spezielle

„base64Binary“ bzw. „string“ Einzeltypen behandelt werden um zu vermeiden dass bei Datenübertragungen pro Byte oder Zeichen ein separates XML-Element anfällt.

- Parameter und Resultate die dagegen einen Mengentyp aufweisen (Arrays, Collections, NICHT byte[] oder char[]) werden dagegen bei Senden von null auf der Empfängerseite immer als leere Menge dargestellt; hier ist es schlicht nicht möglich null zu empfangen!

Die Interface-Halo Klasse **Entity** ist bereits so vorbereitet dass ihre Übertragung generell gelingt (siehe `@XmlAccessorType`). Für die Subklassen von **Entity** (*Article*, *Customer*, *Order* und *OrderItem*) gilt dies nicht, hier sollte noch jede Klasse mit `@XmlRootElement` annotiert werden um sie auch außerhalb von SOAP übertragen zu können, z.B. mittels REST-Services. Zudem muss noch jedes nichtstatische Feld annotiert werden welches übertragen werden soll: Primitive Datentypen und Strings werden dabei typischerweise mittels `@XmlAttribute` annotiert, alle anderen Feldtypen dagegen mittels `@XmlElement`.

**Beachtet dabei:** Auch hier wäre es möglich mittels `@XmlElement` bzw. `@XmlAttribute` weitere Constraints zu definieren um die Nullbarkeit bzw. Kardinalität der Feld-Typen einzuschränken, und dies würde auch hier auf die Typen-Sektion der generierten WSDL Einfluss haben. Dummerweise werden diese Constraints jedoch von den generierten Endpoints und Proxies genauso wie oben bei den Parametern/Resultaten ignoriert. Einzig das Attribut `nilable=true` (von `@XmlElement`) führt zu einer Änderung im Laufzeitverhalten; statt das Feld bei null wegzulassen würde es dann als spezieller XML null-Wert repräsentiert, was jedoch kein echter Gewinn ist.

**Beachtet zudem:** Die abstrakte Halo-Klasse **Entity** ist bereits zusätzlich mit der Annotation `@XmlSeeAlso` versehen, welche `@XmlType` für einen gesamten Klassenbaum impliziert:

- `@XmlType` sorgt dafür dass die Klasse (sowie alle ihre Subklassen) in der generierten WSDL als separater und benannter Typ aufgeführt wird. Ohne eine solche Annotation würde die Marshaling-Struktur der Halo-Klasse(n) in der WSDL pro Verwendung als Parameter oder Resultat einmal komplett aufgelistet, was extrem redundant, und für top-down Clients zudem schwer handhabbar wäre. Daher sollte jede finale Einzel-Klasse stets mit `@XmlType` annotiert werden!
- `@XmlSeeAlso` adressiert gegenüber `@XmlType` zusätzlich **Polymorphismus**: Die Annotation listet alle erlaubten Subklassen einer Klasse auf, welche dann bei Datenübertragungen polymorph behandelt werden sollen. Ohne diese Annotation werden solche Klassen in der generierten WSDL nicht-polymorph gehandhabt, was wie immer in Fällen von fehlendem Polymorphismus schwerwiegendste Auswirkungen hat:
  - Ohne `@XmlSeeAlso` überträgt eine Service-Methode `f(A)` zum Server immer eine Instanz von A, selbst wenn vom Client eine Instanz einer Subklasse von A übergeben wird; dies gilt äquivalent für die Resultate von Service-Methoden!
  - Ohne `@XmlSeeAlso` könnten abstrakte Klassen (oder Interfaces) wie **Entity** im Service-Interface überhaupt nicht als Parameter/Resultate deklariert werden, da auf der Empfängerseite keine Instanz von ihnen erzeugt werden kann!

**Daher sollte stets `@XmlSeeAlso` für jede Halo-Klasse mit Subklassen annotiert werden**, was dann `@XmlType` für ihre Subklassen unnötig macht!

### Aufgabe 3: ShopServer

Erzeugt eine Klasse *ShopServer* welche die *ShopService* und *AutoCloseable* Interfaces implementiert, so dass ein funktionierender Web-Service mit dem Namen „ShopService“ entsteht. Diese Klasse sollte folgende Instanzvariablen besitzen:

- `private final URI serviceURI`
- `private final Endpoint endpoint`
- `private final ShopConnector jdbcConnector`
- `private final double taxRate`

Zudem muss sie analog zu *SoapChatServer* noch mit `@WebService(...)` annotiert werden. Erzeugt einen Konstruktor dazu der die Parameter `servicePort`, `serviceName`, und `taxRate` übernimmt, und erzeugt daraus die Service-URI und den JAX-WS Endpoint wie im *RpcChatServer* Beispiel; den Shop-Connector erzeugt ihr durch einfache Instantiierung ohne Parameter. Die Instanz-Methoden `close()` sowie `getServiceURI()` kopiert ihr aus dem *SoapChatServer* Beispiel, wobei beim Schließen zusätzlich der Shop-Connector geschlossen werden muss. Die statische Methode `main()` kopiert ihr ebenfalls aus dem *SoapChatServer* Beispiel, und passt sie so an dass zusätzlich eine Steuerrate (double, Wertebereich [0.0-1.0]) aus den Argumenten ausgelesen, und bei der Server-Konstruktion übergeben werden kann.

Implementiert darauf aufbauend die Methoden des Service-Interfaces so, dass sie in 1:1 Aufrufen von Methoden des Shop-Connectors resultieren; dieser fungiert also als eine Art Delegat, ähnlich wie im *SoapChatServer* Beispiel. Dabei sollen jedoch zudem lokale ACID Transaktionen zum Einsatz kommen, da es sonst passieren kann dass Teile der notwendigen Datenbank-Updates bereits verbucht sind wenn im Shop-Connector Exceptions geworfen werden, und damit inkonsistente Datenbestände entstehen. Geht daher wie folgt vor:

- Schaltet im *ShopServer* Konstruktor den auto-commit Modus der JDBC-Verbindung des gerade erzeugten Shop-Connectors ab → `this.jdbcConnector.getConnection().setAutoCommit(false)`.
- Sendet in jeder Service-Methode nach dem Methodenaufruf des Shop-Connectors die Nachricht `commit()` an die JDBC-Verbindung desselben. Damit wird die aktuelle ACID-Transaktion committed, i.e. alle vorgenommenen Änderungen am Datenbestand werden persistent, und für andere Transaktionen sichtbar.
- Platziert zudem einen try-Block um alle Befehle der Service-Methode herum, mit einem catch-Block für den Typ `Exception`; sendet der JDBC-Verbindung dort die Nachricht `rollback()` bevor ihr die gefangene `Exception` einfach wieder werft; das Java7 „Precise Rethrow“-Feature nimmt Euch hier eine Menge Arbeit ab weil es dafür sorgt dass die `Exception` je nach Typ korrekt gecastet wieder-geworfen wird.
- Des Weiteren müsst ihr noch mittels passendem `Synchronized-Statement` in jeder Service-Methode verhindern dass die Änderungen durch eine parallel laufende weitere Service-Methode committed oder zurück gerollt werden.

**Hinweis:** Die Notwendigkeit JDBC-Statements sowie Transaktionen zu synchronisieren kann durch Verwendung von sogenannten *Connection-Pools* umgangen werden, wodurch jeder Service-Methode zur Laufzeit ihre „eigene“ JDBC-Verbindung (und damit ein neuer Shop-Connector) zugeteilt würde. Dies würde die Service-Performance massiv steigern, weil dadurch die Notwendigkeit zur Synchronisation der JDBC-Verbindung entfiel, wodurch dann die Service-Methoden parallel ausgeführt werden könnten. In Java-EE ist aus diesem Grund der Einsatz gepoolter JDBC-Verbindungen der Normalfall, und die Architektur ist darauf optimiert diese (und Transaktionen) so einfach wie möglich nutzbar zu machen. In Java-SE würde dies jedoch Einarbeitung und Einsatz von Open-Source Libraries sowie gepoolter JDBC *DataSources* bedingen, und geht daher deutlich über den Umfang dieser Übung hinaus.

## Aufgabe 4: ShopServiceTest

Entwickelt eine Klasse *ShopServiceTest*, welche es erlaubt Euren JAX-WS basierten *ShopService* automatisiert zu testen. Dazu muss Eurem Projekt die Library JUnit (Version 4) hinzugefügt werden.

In JUnit ist ein Test im Prinzip ein beliebiges Java-Objekt, typischerweise stateless. JUnit bietet die Annotation `@Test` um public Methoden zu annotieren die als Testcases ausgeführt werden sollen. Zur Ausführung muss dann die Test-Klasse markiert, und mittels `run as->JUnit Testcase` ausgeführt werden.

Die Annotationen `@Before` und `@After` können verwendet werden um vor/nach jedem (!) Testfall Logik für Initialisierungs- und Aufräum-Operationen auszuführen. Verwendet diese um sowohl im Fehler- als auch Erfolgsfall Euren Datenbankinhalt stets gleich zu halten.

Innerhalb der Testcases wiederum könnt ihr mittels der Facade `org.junit.Assert` Testbedingungen überprüfen lassen. Um das Werfen von Exceptions zu überprüfen könnt ihr im try-Block `Assert.fail()` aufrufen, und im catch-Block die Exception einfach wegfangen.

Startet sodann euren Shop Server und testet ihn mit dem neuen *ShopServiceTest*. Ziel ist es jede Service-Methode mindestens ein mal aufzurufen, und bei den Parametern Grenzfälle zu überprüfen.