2 Kapitel

WIR BAUEN UNS EINE WEB-APPLIKATION (ZWEITER TEIL)

Dieses Manuskript steht allen Teilnehmern der Lehrveranstaltung *Software Engineering* (SE) an der *Technischen Fachhochschule Berlin* als unterrichtsbegleitendes Lehrmaterial frei zur Verfügung. Die Nutzung durch andere Personen oder zu anderen Zwecken bedarf zur Vermeidung möglicher Verletzungen des deutschen Urheberrechts der vorherigen Inkenntnissetzung und Erlaubnis des Autors.

2.5 Persistente Klassen mit JPA (Zweite Fingerübung)

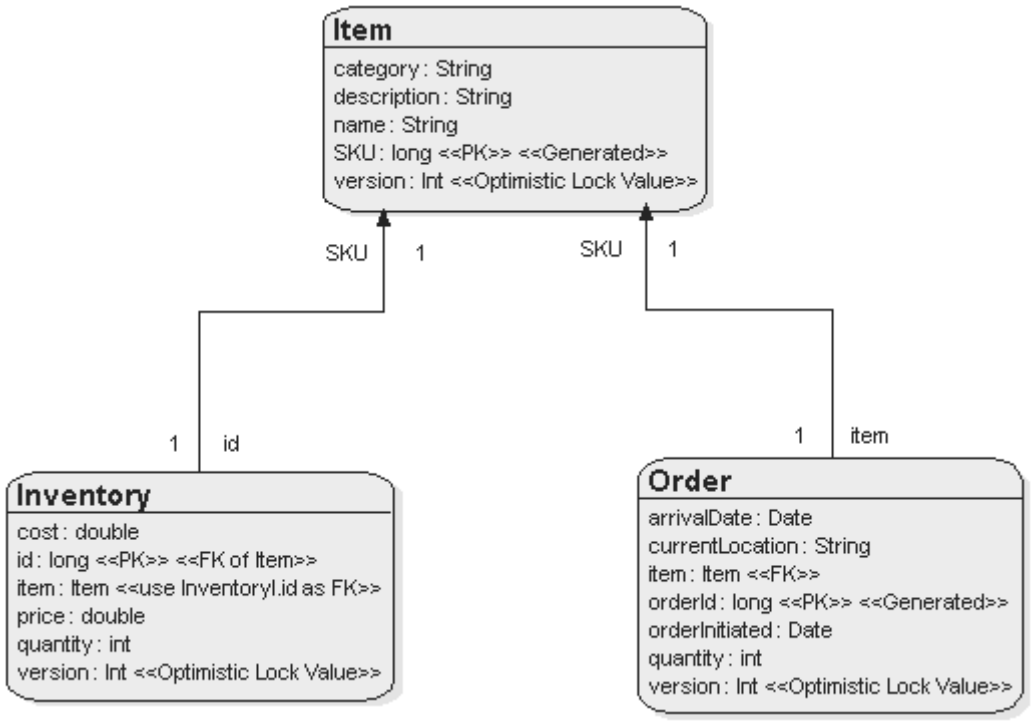
Show

Unsere zweite Fingerübung besteht darin, Java-Objekte (präziser ausgedrückt: die *Attributinhalte* bestimmter Java-Objekte) über die Lebensdauer einer Anwendung hinweg persistent zu speichern. Dazu wird der Java-Quellcode um so genannte Annotationen (*annotations*) erweitert. Annotationen sind textuelle Metadaten, die in die Java-Klasse kompiliert und zur Laufzeit der Anwendung vom JPA-Anbieter *(JPA persistance provider*) ausgewertet werden, um das Persistenz-Verhalten zu ermöglichen. Dem JPA-Anbieter wird durch sie mitgeteilt, welche Klassen überhaupt persistent sein sollen, und um weitere spezifische Persistenz-Eigenschaften dieser Klassen zu bestimmen.

Entity Relationship-Modell

Show

Zugrunde liegt ein sehr einfaches ER-Modell (*Entity Relationship Model*).



Show

Die persistente Klasse Inventory (*@Entity*)

Die Klasse *Inventory*, deren Entität oben im ER-Modell modelliert ist, befindet sich in der Java-Datei *Inventory.java*. Sie ist im Ordner *%TUTORIAL_HOME%**persistence-unit**\src\oracle\toplink\jpa\example\inventory\model* zu finden. Diese Klasse soll künftig persistent sein. Erweitern Sie die Datei um die folgende *@Entity*-Annotation:

```
@Entity
public class Inventory {
  […]
}
```

Standardmäßig ist der Name der Datenbanktabelle automatisch der Name der Klasse, auf die sich die Tabelle bezieht. Dieses Verhalten könnte durch die *@Table*-Annotation beeinflusst werden. Durch die *@Entity*-Annotation werden *alle* Attribute der Klasse persistent, jedoch könnten einzelne Attribute durch eine *@Transient*-Annotation ausgeschlossen werden.

Show

Ein Attribut der Klasse zum Primärschlüssel einer Tabelle ernennen (*@Id*)

Jede Datenbanktabelle, die sich in der zweiten Normalform befindet, besitzt einen Primärschlüssel. Der Primärschlüssel wird durch die *@Id*-Annotation spezifiziert. Erweitern Sie die Java-Datei durch diese Annotation:

```
@Entity
public class Inventory {
  @Id
  protected long id;
  […]
}
```

Show

Die Eigenschaften eines Attributs in der Tabelle definieren (*@Column*)

Standardmäßig korrespondiert der Name und Typ eines Attributs der Datenbanktabelle automatisch mit dem Namen und Typ des entsprechenden Attributs der Java-Klasse. Dieses Verhalten kann durch die *@Column*-Annotation beeinflusst werden:

```
@Entity
public class Inventory {
  @Id
  @Column(name="ITEM_SKU", insertable=false, updatable=false)
  protected long id;
  […]
  public void setItem(Item item) {
    this.item = item;
    this.id = item.getSKU();
  }
  […]
}
```

Da die Tabelle *Inventory* den Inhalt ihres Primärschlüssels *ITEM_SKU* ausschließlich aus der Fremdschlüsselbeziehung auf ihr korrespondierendes *Item* durch die Operation *setItem()* bezieht (siehe auch das zugrunde liegende ER-Modell), muss sichergestellt sein, dass *ITEM_SKU* niemals direkt durch INSERT- oder UPDATE-Statements überschrieben wird. Dies wird durch *insertable=false* und *updatable=false* erreicht.

Show

Die Multiplizität einer Beziehung zwischen Tabellen definieren (*@OneToOne*)

Die Beziehung zwischen den Tabellen *Inventory* und *Item* wird durch die *@OneToOne*-Annotation definiert. Darüber hinaus stehen noch die Annotations *@ManyToMany*, *@ManyToOne* und *@OneToMany* zur Verfügung.

```
@Entity
public class Inventory {
  @Id
  @Column(name="ITEM_SKU", insertable=false, updatable=false)
  protected long id;

  @OneToOne
  protected Item item;
  […]
}
```

Show

Eine Beziehung zwischen Klassen definieren (*@JoinColumn*)

Zwar ist durch die Annotation *@OneToOne* nun datenbankseitig die Beziehung der Tabellen *Inventory* und *Item* definiert, doch möchten wir beim Materialisieren (beim Erstellen) des *Inventory*-Objekts gern den entsprechenden Verweis auf ein Item-Objekt antreffen. Wir erweitern die Klasse deswegen um eine weitere Annotation:

```
@Entity
public class Inventory {
  @Id
  @Column(name="ITEM_SKU", insertable=false, updatable=false)
  protected long id;

  @OneToOne
  @JoinColumn(name="ITEM_SKU")
  protected Item item;
  […]
}
```

Show

Optimistisches Sperren (*@Version*)

Der JPA-Anbieter geht davon aus, dass die Anwendung sich selbst um die Integrität der verwalteten Daten sorgt. Oracle empfiehlt zu diesem Zweck hingegen, die *@Version*-Annotation zu verwenden. Mit ihr wird ein optimistisches Sperren (*optimistic locking*) seitens des JPA-Anbieters aktiviert. Optimistisches Sperren ist ähnlich der Versionsverwaltungsstrategie *Copy-Modify-Merge* (vergleiche hierzu [SE I, 3. VL: Strategien zur Softwareversionsverwaltung]): erst beim Zurückschreiben eines Datensatzes in die Datenbank wird geprüft, ob die seinerzeit ausgelesenen Daten zwischenzeitlich „von jemand anderem“ überschrieben wurden. Das Zurückschreiben wird in diesem Fall abgelehnt, bevor es zu einem Konflikt käme. Die Sperre erfolgt mittels eines dafür zu bestimmenden numerischen Attributs in der Klasse, das den Sperrwert (*lock value*) enthält.

```
@Entity
public class Inventory {
  […]
  @Version
  protected int version;
  […]
}
```

Show

SQL-Statements vordefinieren (*@NamedQuery*)

Zwar kann in einer JPA-Anwendung der *EntityManager* genutzt werden, um SQL-Abfragen dynamisch zur Laufzeit der Anwendung zu erstellen, jedoch können mittels der *@NamedQuery*-Annotation auch im Java-Quellcode bereits SQL-Abfragen auf sehr elegante Weise formuliert werden. Dies bietet sich vor allem für sehr komplexe oder häufig verwendete Abfragen an:

```
@Entity
@NamedQuery(
  name="inventoryForCategory",
  query="SELECT i FROM Inventory i
         WHERE i.item.category = :category
               AND i.quantity <= :maxQuantity"
)
public class Inventory {
  […]
}
```

Das SQL-Statement wurde in diesem Manuskript ausschließlich zur besseren Lesbarkeit in mehrere Zeilen umgebrochen. Achten Sie darauf, in der Datei das gesamte SQL-Statement innerhalb des Annotationsattributs *query="..."* in **einer einzelnen** Zeile unterzubringen, da *ant* anderenfalls beim Übersetzungsvorgang Fehler bringt.

Show

Weitere Ergänzung an anderen Dateien *(@GeneratedValue)*

Auch die Datei *Item.java*, deren Klasse *Item* in hohem Maße mit der eben bearbeiteten Klasse *Inventory* in Beziehung steht, benötigt einige Erweiterungen in Form von Annotationen. Sie ist ebenfalls in dem von Ihnen eben bereits bearbeiteten Ordner *%TUTORIAL_HOME%\persistence-unit\src\oracle\toplink\jpa\example\inventory\model* zu finden:

```
@Entity
public class Item {
  protected long SKU;
  […]
  @Id
  @GeneratedValue
  public long getSKU() {
    return SKU;
  }
  […]
  @Version
  public void setVersion(int Version) {
    this.version = version;
  }
  […]
}
```

Die Getter-Operation *getSKU()* wird als Primärschlüssel definiert. Einen eindeutigen Schlüsselinhalt zu generieren, überlassen wir durch die *@GeneratedValue*-Annotation dem JPA-Anbieter. Generell kann entweder das Attribut selbst annotiert werden (so wie in *Inventory.java* geschehen) oder aber die Getter-Operation, die mit dem Attribut korrespondiert.

Die Setter-Operation *setVersion()* dient in diesem Fall zum optimistischen Sperren (*optimistic locking*).

Show

Letztlich benötigt noch die Datei *Order.java* ein paar Erweiterungen. Sie ist auch im Ordner *%TUTORIAL_HOME%\persistence-unit\src\oracle\toplink\jpa\example\inventory\model* zu finden:

```
@Entity
@Table(name="ORDER_TABLE")
@NamedQueries(
  {
    @NamedQuery(
      name="shippedOrdersForItem",
      query="SELECT o FROM Order o JOIN o.item i
             WHERE i.sKU = :itemId AND o.arrivalDate is not null"
    ),
    @NamedQuery(
      name="pendingOrdersForItem",
      query="SELECT o FROM Order o
             WHERE o.item.sKU = :itemId AND o.arrivalDate is null"
    )
  }
)
public class Order {
  @Id
  @GeneratedValue
  protected long orderId;
  […]
  @Version
  protected int version;
  […]
  @OneToOne
  protected Item item;
  […]
}
```

Die Klasse *Order* kann nicht direkt in eine Tabelle abgeleitet werden, weil das Wort *ORDER* in den meisten relationalen Datenbankverwaltungssystemen bereits reserviert ist. Wir nutzen deshalb die Annotation *@Table*, um einen anderen Tabellennamen anzugeben.

Wenn mehr als eine *NamedQuery* definiert werden soll, müssen diese in einer *@NamedQueries*-Annotation zusammengefasst werden. Achten Sie auch dieses Mal darauf, in der Datei das gesamte SQL-Statement innerhalb des Annotationsattributs *query*="..." in **einer einzelnen** Zeile unterzubringen.

Das Attribute *orderId* wird Primärschlüssel der Tabelle. Die Annotation *@GeneratedValue* teilt dem JPA-Anbieter mit, dieses Attribut selbständig zu verwalten.

Das Attribute *version* dient für die optimistische Sperre.

Das Attribut *item* ist durch die Annotation *@OneToOne* gekennzeichnet, um die Beziehung zwischen den Tabellen *Order* und *Item* zu definieren.

2.6 Die Persistenzeinheit (persistence unit) konfigurieren

Show

Der so genannte Entity Manager ist die Schnittstelle zum JPA-Anbieter, um grundlegende Persistenzoperationen auf der Datenbank ausführen zu können (erstellen von Datensätzen, lesen, aktualisieren und löschen).

Die Persistenzeinheit (verwenden Sie besser den englischen Begriff *Persistence Unit*) definiert die Konfiguration dieses Entity Managers. Folgende Bedingungen müssen erfüllt sein:

- Jede Persistenzeinheit besitzt einen Namen.
- Jede Software-Anwendung besitzt nur eine Persistenzeinheit.

Der Name der Persistenzeinheit wird später verwendet werden, wenn die Entity Manager Factory in unserer Java-Anwendung erstellt wird. Die Persistenzeinheit wird in der Datei *persistence.xml* definiert. Sie befindet sich in unserem Oracle-Beispiel im Ordner *%TUTORIAL_HOME%\persistence-unit\src\META-INF\persistence.xml*. Wir werden Sie nun entsprechend anpassen:

Sämtliche persistenten Klassen müssen darin definiert werden. Im vorangegangenen Abschnitt 2.4 haben wir die Klassen *Inventory*, *Order* und *Item* identifiziert:

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
[…]
<class>oracle.toplink.jpa.example.inventory.model.Inventory</class>
<class>oracle.toplink.jpa.example.inventory.model.Order</class>
<class>oracle.toplink.jpa.example.inventory.model.Item</class>
[…]
</persistence-unit>
```

Show

Ferner ist es erforderlich, die Schnittstelle zum tatsächlich verwendeten relationalen Datenbankverwaltungssystem zu definieren. In unserem Fall ist dies Oracle 10g Express Edition:

```
[…]
<jdbc-driver>oracle.jdbc.OracleDriver</jdbc-driver>
<jdbc-url>jdbc:oracle:thin:@<hostname>:1521:<sid></jdbc-url>
<jdbc-user><username></jdbc-user>
<jdbc-password><password></jdbc-password>
[…]
```

Verwenden Sie als JDBC-Treiber *nicht* die Klasse *oracle.jdbc.Driver.OracleDriver*, obwohl sie in der Literatur bzw. im Internet häufig genannt wird: sie ist *deprecated*.

Verwenden Sie den Namen des Datenbank-Servers statt *<hostname>* und die Datenbank-Identifikation statt *<sid>* (in unserem Beispiel lautet die *XE*).

Verwenden Sie den tatsächlichen Benutzernamen und das Kennwort für Ihre Datenbank statt *<username>* und *<password>*.

2.7 Die Verwendung des Entity Managers

Im Abschnitt 2.4 wurde das OR-Mapping (*object-relational mapping*) zwischen den Java-Klassen und den korrespondierenden Datenbanktabellen vorgenommen. Im Abschnitt 2.5 haben wir den JPA-Anbieter unseren Wünschen entsprechend konfiguriert.

Im folgenden werden wir uns mit der so genannten Entity Manager Factory des JPA-Anbieters beschäftigen, mit ihr einen Entity Manager erstellen und mit seiner Hilfe Datensätze in der Datenbank erstellen, auslesen, aktualisieren und löschen.

Die Klasse, welche die grafische Benutzungsoberfläche unserer Web-Applikation realisiert, heißt *oracle.toplink.jpa.example.inventory.ui.InventoryManagerBean*. Sie enthält Referenzen auf zwei weitere Klassen, die JPA verwenden, und uns interessieren:

Die Klasse *oracle.toplink.jpa.example.inventory.services.impl.ManagedOrderBean* (implementiert *oracle.toplink.jpa.example.inventory.services.OrderService*).

Die Klasse *oracle.toplink.jpa.example.inventory.services.impl.ManagedInventoryBean* (implementiert *oracle.toplink.jpa.example.inventory.services.InventoryService*).

Die beiden blau markierten Klassen verwenden ein Objekt der Hilfsklasse *oracle.toplink.jpa.example.inventory.services.impl.JPAResourceBean*, um ein Entity Manager Factory-Objekt der Persistenzeinheit *default* (siehe Abschnitt 2.5) zu erstellen, die wir in der Datei *persistence.xml* konfiguriert haben.

Show

Die Entity Manager Factory erstellen, um den Entity Manager zu nutzen

Ist die Entity Manager Factory erstellt, können die Klassen *ManagedOrderBean* und *ManagedInventoryBean* diese verwenden, um einen Entity Manager zu erhalten, mit dessen Hilfe sie die grundlegenden Datenbankoperationen durchführen können. Die Hilfsklasse *JPAResourceBean* stellt folgende Operation *getEMF()* zur Verfügung:

```
public EntityManagerFactory getEMF() {
  if(emf == null) {
    emf = Persistence.createEntityManagerFactory("default");
  }
  return emf;
}
```

Alle weiteren Datenbankoperationen basieren auf der *EntityManagerFactory*.

Show

Einen neuen Datensatz einer Tabelle erstellen

Öffnen Sie für das weitere Vorgehen im Beispiel die Java-Datei *%TUTORIAL_HOME%**web-application**\src\oracle\toplink\jpa\example\inventory\services\impl\ManagedOrder Bean.java*.

In der Klasse *ManagedOrderBean* wird ein Datensatz in der Tabelle *Order* erstellt und mit dem Inhalt eines der Operation *createNewOrder()* übergebenen Objekts *order* gefüllt:

```
public void createNewOrder(Order order) {
  EntityManager em = jpaResourceBean.getEMF().createEntityManager();
  try {
    em.getTransaction().begin();
    em.persist(order);
    em.getTransaction().commit();
  }
  finally {
    em.close();
  }
}
```

Show

Den Datensatz einer Tabelle lesen

Die folgende Operation *getOrderById()* zeigt, wie die Klasse *ManagedOrderBean* den Entity Manager nutzt, um einen existierenden Datensatz der Tabelle *Order* mit einem bestimmten Primärschlüssel-Inhalt *orderId* anzufordern. Die Operation *find()* liefert ein vollständiges *Order*-Objekt zurück:

```
public Order getOrderById(long orderId) {
  EntityManager em = jpaResourceBean.getEMF().createEntityManager();
  try {
    return em.find(Order.class, orderId);
  }
  finally {
    em.close();
  }
}
```

Show

Den Datensatz einer Tabelle aktualisieren

Die Operation *alterOrderQuantity()* zeigt, wie die Klasse *ManagedOrderBean* den Entity Manager nutzt, um einen existierenden Datensatz der Tabelle *Order* zu aktualisieren. Die Änderung wird in die Datenbank zurück geschrieben, sobald die lokale Transaktion bestätigt (*commited*) wird:

```
public void alterOrderQuantity(long orderId, int newQuantity) {
  EntityManager em = jpaResourceBean.getEMF().createEntityManager();
  try {
    em.getTransaction().begin();
    Order order = em.find(Order.class, orderId);
    order.setQuantity(newQuantity);
    em.getTransaction().commit();
  }
  finally {
    em.close();
  }
}
```

Show

Den Datensatz einer Tabelle löschen

Das letzte Beispiel zeigt, wie die Klasse *ManagedOrderBean* ihren Entity Manager nutzt, um einen Datensatz der Tabelle *Order* zu entfernen. Die Änderung wird in die Datenbank zurück geschrieben, sobald die lokale Transaktion bestätigt (*commited*) wird:

```
public void requestCancelOrder(long orderId) {
  EntityManager em = jpaResourceBean.getEMF().createEntityManager();
  try {
    em.getTransaction().begin();
    Order order = em.find(Order.class, orderId);
    em.remove(order);
    em.getTransaction().commit();
  }
  finally {
    em.close();
  }
}
```

2.8 SQL-Abfragen nutzen

Die beiden Klassen *ManagedInventoryBean* und *ManagedOrderBean* nutzen verschiedene SQL-Abfragen, die JPA in Form so genannter JPA-Queries zur Verfügung stellt.

Öffnen Sie für das weitere Vorgehen im Beispiel die Java-Datei *%TUTORIAL_HOME%**web-application**\src\oracle\toplink\jpa\example\inventory\services\impl\ManagedInventory Bean.java*.

Show

Eine Named Query nutzen

Die Operation *createNamedQuery()* des Entity Managers wird genutzt, eine Query-Instanz der im Abschnitt 2.4 definierten Named Query *inventoryForCategory* zurückzuliefern:

```
public class ManagedInventoryBean implements InventoryService {
  […]
  public Collection<Inventory> getInventoryForCategoryMaxQuantity(
                                              String category, int quantity) {
    EntityManager em = jpaResourceBean.getEMF().createEntityManager();
    try {
      Query query = em.createNamedQuery("inventoryForCategory");
      query.setParameter("category", category);
      query.setParameter("maxQuantity", quantity);
      return query.getResultList();
    }
    finally {
      em.close();
    }
  }
  […]
}
```

Show

Eine dynamische Abfrage nutzen

Die Operation *createQuery()* des Entity Managers wird genutzt, eine dynamische Query-Instanz in der Klasse *ManagedInventoryBean* zu erzeugen:

```
public class ManagedInventoryBean implements InventoryService {
  […]
  public collection<Category> getCategories() {
    EntityManager em = jpaResourceBean.getEMF().createEntityManager();
    try {
      String sql = "SELECT new oracle.toplink.jpa.example.inventory.
                    Nonentity.Category(i.category)
                    FROM Item i GROUP BY i.category";
      Collection<Category> result = em.createQuery(sql).getResultList();
      return result;
    }
    finally {
      em.close();
    }
  }
  […]
}
```

Beachten Sie, dass diese SQL-Abfrage eine *Collection* von Objekten des Typs *Category* zurückliefert, die wir gar nicht als persistente Klasse definiert haben („*that are not entity classes*“). Der JPA-Anbieter nutzt dazu die Informationen, die ihm aufgrund des Java-Kompilats zur Verfügung stehen, um diese Hilfsklasse zu instanziieren („*it uses summary data to create this non-entity helper class*“).

Weitere Ergänzungen an anderen Dateien

Öffnen Sie für das weitere Vorgehen im Beispiel die Java-Datei *%TUTORIAL_HOME%\web-application\src\oracle\toplink\jpa\example\inventory\services\impl\ManagedOrderBean.java*:

```
public class ManagedOrderBean implements OrderService {
  […]
  public Collection<Order> getShippedOrdersForItem(long itemId) {
    EntityManager em = jpaResourceBean.getEMF().createEntityManager();
    try {
      Query query = em.createNamedQuery("shippedOrdersForItem");
      query.setParameter("itemId", itemId);
      return query.getResultList();
    }
    finally {
      em.close();
    }
  }

  public Collection<Order> getPendingOrdersForItem(long itemId) {
    EntityManager em = jpaResourceBean.getEMF().createEntityManager();
    try {
      Query query = em.createNamedQuery("pendingOrdersForItem");
      query.setParameter("itemId", itemId);
      return query.getResultList();
    }
    finally {
      em.close();
    }
  }
  […]
}
```

Interpretieren Sie etwaige Fehlermeldungen des Java-Compilers, und ergänzen Sie analog alle weiteren JPA-Operationen, wo dies notwendig ist.

2.9 Überprüfung der Web-Applikation

Übersetzen Sie nun die Java-Anwendung. Wechseln Sie dazu nach *%TUTORIAL_HOME%* und führen Sie das folgende Kommando in einer Kommando-Shell aus:

```
ant -f build.xml package.webapp
```

Kopieren Sie die durch diesen Paketierungs- und Veröffentlichungsvorgang entstandene Datei *jpa-example.war* aus dem Ordner *%TUTORIAL_HOME%\web-application\deploy* in den Ordner *%CATALINA_HOME%\webapps*.

Geben Sie nun in Ihrem Lieblings-Browser die folgende Adresse ein:

```
http://<hostname>:8081/jpa-example/index.jsp
```

Ersetzen Sie <*hostname*> durch den Namen des Tomcat-Servers. Öffnet sich eine ähnliche Seite wie bereits im Manuskript-Teil [SE II, 2. VL: Wir bauen uns eine Web-Applikation, Abschnitt 2.2 - Überprüfung der Installationen], dann haben Sie die Implementierung korrekt durchgeführt. Starten Sie bei Bedarf den Dienst des Tomcat-Servers neu.

Bitte führen Sie diese gesamte zweite Fingerübung im Rahmen einer Übungsveranstaltung der Lehrveranstaltung *Software Engineering II* durch.