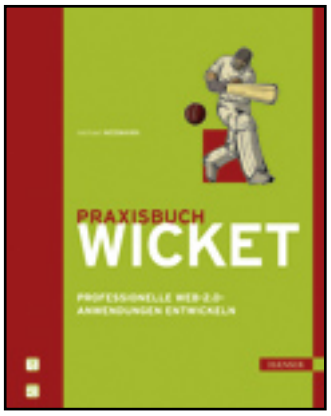
HANSER



Leseprobe

Michael Mosmann

Praxisbuch Wicket

Professionelle Web-2.0-Anwendungen entwickeln

ISBN: 978-3-446-41909-4

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41909-4

sowie im Buchhandel.

3

3 Mit Leben füllen

Nachdem wir nun die Grundstruktur der Anwendung erstellt und alle Teilprojekte mit allen Abhängigkeiten definiert haben, füllen wir das Ganze mit Inhalt.

3.1 Konfiguration mit Spring

Wie in Abschnitt 1.3.1.2 beschrieben, greifen wir für die Konfiguration der Anwendung auf das Spring-Framework zurück. Dazu erstellen wir in den verschiedenen Schichten die notwendigen Konfigurationsdateien und verknüpfen diese zu einer großen Anwendungskonfiguration.

Die Konfiguration wird durch XML-Dateien beschrieben. In diesen Dateien werden Objekte (`<bean>`) definiert, indem einem Objekt eine ID zugewiesen wird und die Attribute des Objekts mit definierten Werten belegt werden. Die verschiedenen Objekte können dann über die ID verknüpft werden, indem z.B. das Attribut des einen Objekts mit einem anderen Objekt belegt wird. Für ein besseres Verständnis sollte man sich unbedingt die Dokumentation[1] des Frameworks ansehen.

Das Spring-Framework (kurz: Spring) stellt neben dem reinen Dependency Injection-Mechanismus außerdem noch Hilfsfunktionen bereit, die über entsprechende XML-Tags benutzt werden können. Dazu muss man die notwendigen Definitionen in der Konfigurationsdatei einbinden. In der Dokumentation des Frameworks gibt es einen Abschnitt, der diesem Thema gewidmet ist (XML-Schema based configuration[2]).

[1] http://www.springsource.org/documentation
[2] http://static.springframework.org/spring/docs/2.0.x/reference/xsd-config.html

3.2 Datenbankkonfiguration

In allen Datenbankkonfigurationsteilprojekten fügen wir passende Konfigurationsdateien unterhalb des Pfades `src/main/resources/de/wicketpraxis/db/config/` hinzu. In unserem Beispielprojekt benutzen wir drei verschiedene Datenbankkonfigurationen:

- Die erste Datenbank ist eine Datenbank auf dem Entwicklungsrechner oder die Produktivdatenbank, da in beiden Fällen derselbe Datenbanktreiber und dieselbe Einstellung benutzt wird.
- Die zweite Datenbank ist eine InMemory-Datenbank, die nur für die Unit-Tests benutzt und nach dem Beenden der Tests gelöscht wird.
- Die dritte Datenbank entspricht der ersten Datenbank, allerdings mit veränderten Einstellungen. Die Anpassungen veranlassen Hibernate, das Datenbankschema an die Struktur unserer Persistenzschicht anzupassen. Dazu werden die notwendigen Tabellen erzeugt und datenbankabhängige Optimierungen vorgenommen.

3.2.1 Teilprojekt dbconfig

Wir erstellen in dem oben erwähnten Pfad die Konfiguration für die Produktivdatenbank. Diese Einstellung wird auch benutzt, wenn die Anwendung lokal ausgeführt wird.

Listing 3.1 dbconfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-2.0.xsd">

  <bean id="dataSource" \
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/wicketpraxis"/>
    <property name="username" value="wicketpraxis"/>
    <property name="password" value="wicketpraxis"/>
  </bean>

  <util:map id="hibernateProperties" map-class="java.util.Properties" \
    key-type="java.lang.String" value-type="java.lang.String">
    <entry key="hibernate.dialect" \
       value="org.hibernate.dialect.MySQL5InnoDBDialect"></entry>
    <entry key="hibernate.connection.pool_size" value="10"></entry>
    <entry key="hibernate.statement_cache.size"  value="10"></entry>
  </util:map>
</beans>
```

Es werden zwei Objekte definiert:

- Das Objekt `datasource` konfiguriert den JDBC-Datenbankzugriff. Dafür sind neben der Datenbanktreiberklasse das Login, das Passwort, der Servername sowie die JDBC-URL anzugeben.

- Im Objekt `hibernateProperties` werden Attribute gesetzt, die von Hibernate ausgewertet werden. Durch das Setzen des Attributs `hibernate.dialect` kann Hibernate auf einen Tabellentyp zurückgreifen, der Transaktionen unterstützt.

3.2.2 Teilprojekt dbconfig-test

Wie eben für das Teilprojekt `dbconfig` erstellen wir im oben erwähnten Pfad, aber im Teilprojekt `dbconfig-test` die Datei `dbconfig-test.xml` mit einem an `dbconfig.xml` angelehnten Inhalt.

Listing 3.2 dbconfig-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >

  <bean id="dataSource" \
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:test"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <util:map id="hibernateProperties" map-class="java.util.Properties" \
    key-type="java.lang.String" value-type="java.lang.String">
    <entry key="hibernate.dialect" \
      value="org.hibernate.dialect.HSQLDialect"></entry>
    <entry key="hibernate.connection.pool_size" value="10"></entry>
    <entry key="hibernate.statement_cache.size"  value="10"></entry>
    <entry key="hibernate.hbm2ddl.auto" value="create-drop"></entry>
    <entry key="hibernate.bytecode.use_reflection_optimizer" \
      value="false"></entry>
    <entry key="hibernate.show_sql" value="true"></entry>
    <entry key="hibernate.format_sql" value="true"></entry>
  </util:map>
</beans>
```

Damit wird für die Unit-Tests eine InMemory-HSQL-Datenbank konfiguriert. Außerdem wird Hibernate so konfiguriert, dass vor Beginn des Tests das Datenbankschema automatisch erzeugt und nach dem Test wieder automatisch gelöscht wird. So findet jeder Test die Datenbank immer im selben Zustand vor.

Hilfreich ist außerdem, dass die resultierenden SQL-Abfragen ausgegeben werden. Wem die umfangreiche Ausgabe zu viel ist, kann die formatierte Ausgabe (`format_sql=false`) auch abschalten. Das Debuggen der Anwendung gestaltet sich einfacher, wenn der Reflection-Optimizer deaktiviert ist.

3.2.3 Teilprojekt dbconfig-schema-update

Wie wir in der letzten Konfiguration gesehen haben, kann Hibernate das Schema einer Datenbank automatisch erzeugen. Das machen wir uns zunutze, wenn wir jetzt eine Konfiguration erstellen, mit der wir das Erzeugen des Datenbank-Schemas erzwingen können.

Wir legen die Datei `dbconfig-schema-update.xml` an, die sich ebenfalls an `dbconfig.xml` orientiert.

Listing 3.3 dbconfig-schema-update.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >

  <bean id="dataSource" \
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/wicketpraxis"/>
    <property name="username" value="wicketpraxis_root"/>
    <property name="password" value="wicketpraxis_root"/>
  </bean>

  <util:map id="hibernateProperties" map-class="java.util.Properties" \
    key-type="java.lang.String" value-type="java.lang.String">
    <entry key="hibernate.dialect" \
      value="org.hibernate.dialect.MySQL5InnoDBDialect"></entry>
    <entry key="hibernate.hbm2ddl.auto" value="update"></entry>
    <entry key="hibernate.connection.pool_size" value="10"></entry>
  </util:map>
</beans>
```

Wie bei `dbconfig-test.xml` wird ein Abgleich zwischen geforderter und vorhandener Datenbankstruktur durchgeführt. Allerdings werden die alten Daten dabei nicht gelöscht (`hibernate.hbm2ddl.auto=update`), sondern nur angepasst.

Dieses Vorgehen kann immer dann zu Problemen führen, wenn man Attribute (und damit den resultierenden Spaltennamen der Tabelle) umbenannt hat. Dann wird eine neue Spalte erzeugt. Die alte Spalte wird jedoch nicht gelöscht. In solchen Fällen sollte man die Datenstruktur nachträglich von Hand anpassen.

Es ist auf diese Weise natürlich ohne Weiteres möglich, auch das Schema einer Produktivdatenbank automatisch anpassen zu lassen. Das Risiko besteht dann allerdings darin, dass das Produktivsystem dadurch in einen unbrauchbaren Zustand versetzt wird und die Anwendung nicht mehr funktioniert. Daher empfehle ich, das Ändern des Schemas der Produktivdatenbank immer von Hand oder durch die Unterstützung von darauf spezialisierten Anwendungen durchzuführen.

3.2.4 Schemagenerierung mit Hibernate

Folgende Werte sind für das Attribut `hibernate.hbm2dll.auto` möglich:

Wert	Bedeutung
validate	prüft die Gültigkeit der Datenbanktabellen
update	erstellt und aktualisiert die Datenbanktabellen
create	erstellt die Datenbanktabellen initial
create-drop	erstellt die Datenbanktabelle beim Start und löscht sie am Ende

Für die Testdatenbank benutzen wir daher am besten `create-drop`. Für das Schema-Update auf die lokale Entwicklungsdatenbank benutzen wir `update`. Für den normalen Zugriff würde sich die Verwendung von `validate` anbieten. Es empfiehlt sich aber, auf die Angabe vollständig zu verzichten, weil Datenbanken ihrerseits die gewünschten Typen

selbst noch einmal anpassen. Damit passt das von Hibernate erwartete Ergebnis nicht zu dem von der Datenbank zurückgemeldeten, was dann einen Fehler in der Anwendung hervorruft.

3.3 Persistenz

Nachdem wir die Datenbankkonfigurationen erstellt haben, können wir uns mit dem Datenbankzugriff beschäftigen. Damit wir das Rad nicht jedes Mal neu erfinden, erstellen wir ein paar Hilfsklassen, die allen abgeleiteten Klassen häufig benutzte Funktionen zur Verfügung stellt. Dazu wechseln wir in das entsprechende Teilprojekt `persistence`.

3.3.1 Datenbankzugriff – Allgemeine Schnittstellendefinition

Alle Datenbankobjekte (Do), also Objekte, die durch die Persistenzschicht in Tabellen abgelegt werden, müssen folgendes Interface implementieren:

Listing 3.4 DoInterface.java

```
package de.wicketpraxis.persistence;

import java.io.Serializable;

public interface DoInterface<K extends Serializable> extends Serializable
{
  public K getId();
}
```

Dadurch ist es möglich, die eindeutige Identifikation für ein Datenbankobjekt zu ermitteln, ohne die genaue Klasse des Objektes zu kennen. Die Datenbankzugriffsklasse (Data Access Object, DAO) muss ebenfalls ein entsprechendes Interface implementieren:

Listing 3.5 DaoInterface.java

```
package de.wicketpraxis.persistence;

...

public interface DaoInterface<K extends Serializable,T extends DoInterface<K>>
{
  void delete(T o);
  void save(T o);
  void update(T o);

  T get(K id);
  T getNew();

  public List<T> findAll(int offset,int max);
  public int countAll();
}
```

Wenn dieses Interface vollständig implementiert ist, kann man über diese Schnittstelle neue Datenbankobjekte erstellen, diese speichern, verändern, über eine ID suchen und letztendlich auch löschen. Außerdem können alle Objekte in einer Liste dargestellt werden.

3.3.2 Datenbankzugriff – Hilfsklassen

Der Datenbankzugriff, der über die eben definierten Schnittstellen möglich ist, muss nicht für jede Klasse neu implementiert werden. Es bietet sich an, diese Funktionen in eine abstrakte Klasse auszulagern, von der dann die konkreten Klassen abgeleitet werden. Außerdem können wir an dieser Stelle die Integration mit Hibernate über Spring realisieren, sodass in der abgeleiteten Klasse nur noch sehr wenige Anpassungen notwendig sind.

Listing 3.6 AbstractDao.java

```
package de.wicketpraxis.persistence;

...
import org.hibernate.*;
import org.springframework.transaction.annotation.Transactional;

public abstract class AbstractDao<K extends Serializable,T extends DoInterface<K>>
implements DaoInterface<K,T>
{
  private static final Logger _logger = \
    LoggerFactory.getLogger(AbstractDao.class);

  private Class<T> _domainClass;
  private SessionFactory _sessionFactory;

  protected AbstractDao(Class<T> domainClass)
  {
    _domainClass=domainClass;
  }

  public SessionFactory getSessionFactory()
  {
    return _sessionFactory;
  }

  public void setSessionFactory(SessionFactory sessionFactory)
  {
    _sessionFactory = sessionFactory;
  }

  public Session getSession()
  {
    return _sessionFactory.getCurrentSession();
  }

  protected Class<T> getDomainClass()
  {
    return _domainClass;
  }

  protected Criteria getCriteria()
  {
    return getSession().createCriteria(_domainClass);
  }

  @Transactional
  public void delete(T object)
  {
    getSession().delete(object);
  }

  @Transactional
  public void save(T object)
  {
    getSession().save(object);
  }
```

```
    @Transactional
    public void update(T object)
    {
      getSession().update(object);
    }

    public T get(K id)
    {
      return (T) getSession().get(_domainClass, id);
    }

    public T getNew()
    {
      try
      {
        return _domainClass.newInstance();
      }
      catch (InstantiationException e)
      {
        _logger.error("newInstance failed",e);
      }
      catch (IllegalAccessException e)
      {
        _logger.error("newInstance failed",e);
      }
      return null;
    }

    public List<T> findAll(int offset,int max)
    {
      Criteria criteria = getSession().createCriteria(_domainClass);
      if (offset!=0) criteria.setFirstResult(offset);
      if (max!=0) criteria.setMaxResults(max);
      return (List<T>) criteria.list();
    }

    public int countAll()
    {
      Criteria criteria = getSession().createCriteria(_domainClass);
      criteria.setProjection(Projections.rowCount());
      return (Integer) criteria.uniqueResult();
    }
  }
```

Wie man sieht, konnten alle Funktionen aus der Interface-Definition umgesetzt werden. Wichtig ist es, darauf hinzuweisen, dass die Annotation `@Transactional` für die Transaktionssteuerung benutzt wird. Spring kümmert sich bei einer so markierten Methode darum, dass vor dem Aufruf eine Transaktion gestartet und am Ende der Methode geschlossen wird.

Hinweis

In abgeleiteten Klassen muss die Annotation `@Transactional` neu gesetzt werden, da Annotationen nicht vererbt werden.

3.3.3 Datenbankzugriff – User

Für unsere Beispielanwendung greifen wir auf eine sehr einfach gehaltene Nutzerdatenbank zurück.

Listing 3.7 User.java

```
package de.wicketpraxis.persistence.beans;

import javax.persistence.*;
...

@Entity
@Table(name="Users")
public class User implements DoInterface<Integer>
{
  Integer _id;
  String _eMail;
  String _name;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public Integer getId()
  {
    return _id;
  }

  public void setId(Integer id)
  {
    _id=id;
  }

  @Column(nullable=false,unique=true,name="email")
  public String getEMail()
  {
    return _eMail;
  }

  public void setEMail(String mail)
  {
    _eMail = mail;
  }

  @Column(nullable=false)
  public String getName()
  {
    return _name;
  }

  public void setName(String name)
  {
    _name = name;
  }
}
```

Die Klasse ist jetzt noch sehr übersichtlich. Wer bereits weitere Attribute identifiziert hat, kann diese einfach hinzufügen. Der Zugriff auf die Nutzer erfolgt dann über eine eigene Klasse. Diese Klasse leitet von `AbstractDao` ab und erbt somit alle grundlegenden Datenbankfunktionen. Wir fügen in diesem Fall noch eine Methode hinzu, die einen Datensatz anhand der E-Mail ermittelt und als Ergebnis zurückgibt.

Listing 3.8 UserDao.java

```
package de.wicketpraxis.persistence.dao;

import org.hibernate.criterion.*;
...

public class UserDao extends AbstractDao<Integer, User>
{
  public static final String BEAN_ID="userDao";
```

```
  public UserDao()
  {
    super(User.class);
  }

  public User getByEMail(String email)
  {
    return (User) getCriteria().add(Property.forName("EMail").eq(email))
     .uniqueResult();
  }
}
```

Damit haben wir alle Klassen zusammen, die für den Datenbankzugriff verantwortlich sind. Jetzt müssen wir die Persistenzschicht noch richtig konfigurieren.

3.3.4 Datenbankzugriff – Konfiguration

Wir legen eine Konfigurationsdatei im Verzeichnis `src/main/resources/de/wicket-praxis/persistence/` an.

Listing 3.9 bean-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

  <bean id="sessionFactory" class="org.springframework.orm.hibernate3. \
    annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="annotatedClasses">
      <list>
        <value>de.wicketpraxis.persistence.beans.User</value>
      </list>
    </property>
    <property name="hibernateProperties" \
      ref="hibernateProperties"></property>
  </bean>

  <bean id="transactionManager" \
    class="org.springframework.orm.hibernate3. \
    HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"></property>
  </bean>

  <tx:annotation-driven transaction-manager="transactionManager" \
    proxy-target-class="true"/>

  <bean id="userDao" class="de.wicketpraxis.persistence.dao.UserDao">
    <property name="sessionFactory" ref="sessionFactory"></property>
  </bean>
</beans>
```

Die von Spring bereitgestellte Session Factory (`sessionFactory`) wertet die Annotationen in der User-Klasse aus. An diese Session Factory, die für das Erzeugen einer Hibernate-Session zuständig ist, wird ein Transaktionsmanager angebunden. Die Anweisung `<tx:annotation-driven>` veranlasst Spring dazu, die Annotationen in den Zugriffsklassen auszuwerten und für die saubere Transaktionsbehandlung Proxy-Klassen für die

Zugriffsklassen bereitzustellen. Das Attribut `proxy-target-class=true` zwingt Spring dabei, auf die Verwendung von dynamischen Proxies zu verzichten und Proxies mithilfe des `cglib` Frameworks bereitzustellen. Dadurch ist es nicht mehr notwendig, ein spezielles Interface pro Klasse zu implementieren. Das erzeugte Zwischenobjekt ruft die Funktionen der eigentlichen Klasse auf und führt dabei vorher und nachher die notwendigen Transaktionsfunktionen aus.

Wir haben jetzt die Persistenzschicht konfiguriert, aber noch keine Datenbankschnittstelle angebunden. Dazu erstellen wir eine weitere Konfigurationsdatei im selben Verzeichnis.

Listing 3.10 persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <import resource="classpath:/de/wicketpraxis/db/config/dbconfig.xml"/>

  <import resource="bean-config.xml"/>

</beans>
```

In dieser Datei greifen wir auf die Datenbankkonfiguration für die Produktivdatenbank zu und referenzieren die Konfiguration der Persistenzschicht.

3.3.5 Persistenz-Tests

Wie bereits erwähnt, kann man mit Hibernate relativ einfach Unit-Tests mit einer temporären Datenbank durchführen. Für die Tests legen wir eine entsprechende Konfiguration an. Diese Konfigurationsdatei legen wir aber im Verzeichnis `src/test/resources/de/wicketpraxis/persistence/` ab, damit diese nur für die Tests benutzt wird und nicht im fertigen Projektarchiv auftaucht.

Listing 3.11 persistence-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <import resource="classpath:/de/wicketpraxis/db/ \
    config/dbconfig-test.xml"/>

  <import resource="bean-config.xml"/>

</beans>
```

Wir benutzen dieselbe `bean-config.xml` jetzt mit der Datenbankkonfiguration für die Testdatenbank. In den Unit-Tests müssen wir daher die Konfigurationsdatei `persistence-test.xml` verwenden. Als Basisklasse für alle weiteren Tests leiten wir eine eigene Klasse von einer Unit-Test-Hilfsklasse des Spring-Frameworks ab. Wir erweitern die Klasse ein wenig und leiten alle konkreten Testklassen von dieser Klasse ab:

Listing 3.12 AbstractTest.java

```
package de.wicketpraxis.persistence;

import org.springframework.test. \
```

```
  AbstractTransactionalDataSourceSpringContextTests;

public abstract class AbstractTest \
  extends AbstractTransactionalDataSourceSpringContextTests
{
  @Override
  protected String[] getConfigLocations()
  {
    return new String[] \
      { "classpath:/de/wicketpraxis/persistence/persistence-test.xml" };
  }

  protected <T> T getBean(String name,Class<T> requiredType)
  {
    return (T) getApplicationContext().getBean(name, requiredType);
  }

  protected <T> T getBean(Class<T> requiredType)
  {
    Map beansOfType = getApplicationContext(). \
      getBeansOfType(requiredType);

    if (beansOfType.size()==1)
    {
      return (T) new ArrayList(beansOfType.values()).get(0);
    }
    return null;
  }
}
```

Der Unit-Test für die User-Klasse gestaltet sich recht übersichtlich. Das liegt sicher auch daran, dass wir bisher nicht allzu viele Funktionen realisiert haben.

Listing 3.13 TestUserDao.java

```
package de.wicketpraxis.persistence.dao;

import junit.framework.Assert;
...

public class TestUserDao extends AbstractTest
{
  public void testKeinNutzer()
  {
    UserDao userDao = getBean(UserDao.BEAN_ID,UserDao.class);
    User user = userDao.get(1);
    Assert.assertNull("Kein Nutzer",user);
  }

  public void testEinNutzer()
  {
    UserDao userDao = getBean(UserDao.BEAN_ID,UserDao.class);

    User nutzer=new User();
    String email = "klaus@test.de";
    nutzer.setEMail(email);
    nutzer.setName("Klaus");
    userDao.save(nutzer);

    nutzer=userDao.get(nutzer.getId());
    Assert.assertNotNull("User",nutzer);
    Assert.assertEquals("Email",email,nutzer.getEMail());

    nutzer=userDao.getByEMail(email);
    Assert.assertNotNull("User",nutzer);
    Assert.assertEquals("Email",email,nutzer.getEMail());
  }
}
```

Der erste Test prüft, ob ein Nutzer mit der ID=1 bereits in der Datenbank ist. Das sollte nicht der Fall sein, weil bei jedem Test die Datenbank wieder gelöscht wird. Im zweiten Test wird geprüft, ob der Nutzer angelegt, gelesen und anhand der E-Mail gefunden werden kann. Diesen Test kann man mit `mvn test` ausführen. Wenn keine Fehler aufgetreten sind, kann man dieses Teilprojekt mit `mvn install` erstellen lassen. Die Unit-Tests werden dabei jedes Mal ausgeführt. Als einfacher Test, ob die Unit-Tests tatsächlich ausgeführt werden, kann man z.B. den Namen in `setName` auf einen anderen Wert ändern. Der Test sollte dann fehlschlagen.

3.3.6 Schema-Update

Hibernate prüft je nach Einstellung vor dem ersten Datenbankzugriff, ob an der in der Datenbank vorhandenen Tabellenstruktur Anpassungen notwendig sind. Je nach Einstellung führt Hibernate dann die entsprechenden Anpassungen durch. Gerade wer wenig Erfahrungen in Bezug auf Datenbanken hat, ist meist besser damit beraten, auf diese Funktion von Hibernate zurückzugreifen. Das Erzeugen und Ändern von Tabellen und Spalten sollte aber nicht jedem Datenbanknutzer erlaubt sein. Daher empfiehlt es sich, die Schema-Update-Funktion nicht automatisch auszuführen, und dafür einen gesonderten Datenbanknutzer mit erweiterten Rechten zu nehmen. Diese Einstellungen haben wir in die `dbconfig-schema-update.xml` einfließen lassen.

Um die Funktion auszuführen, legen wir erst eine eigene Konfigurationsdatei im Verzeichnis `src/test/resources/de/wicketpraxis/persistence/` an.

Listing 3.14 persistence-schema-update.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <import resource="classpath:/de/wicketpraxis/db/ \
    config/dbconfig-schema-update.xml"/>

  <import resource="bean-config.xml"/>

</beans>
```

Diese Datei unterscheidet sich von `persistence-test.xml` nur an der Stelle, an der wir die Datenbankkonfiguration auswählen. Wir benutzen für den Start von Hibernate mit der entsprechenden Konfiguration dasselbe Verfahren wie bei den Persistenz-Tests. Damit dieser „Test“ aber nur bei Bedarf ausgeführt wird, sollte im Klassenname das Wort „Test“ nicht vorkommen.

Listing 3.15 SchemaUpdate.java

```
package de.wicketpraxis.persistence;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SchemaUpdate extends AbstractTest
{
  private static final Logger _logger =
    LoggerFactory.getLogger(SchemaUpdate.class);
```

```
  @Override
  protected String[] getConfigLocations()
  {
    return new String[]{ \
      "classpath:/de/wicketpraxis/ \
        persistence/persistence-schema-update.xml" };
  }

  public void testSchemaUpdate()
  {
    _logger.error("Schema Update");
  }
}
```

Die Testfunktion hat dann eigentlich keine Funktion. Im Vorfeld wird Hibernate mit der referenzierten Datenbankkonfiguration gestartet und das Schema angepasst. Um den Test und damit das Schema-Update zu starten, rufen wir Maven wie folgt auf:

```
$mvn -Dtest=de/wicketpraxis/persistence/SchemaUpdate test
```

Das Teilprojekt ist jetzt soweit fertig. Wir haben durch Datenbanktests die Funktion getestet, wir können per Schema-Update die Datenbanktabellen anlegen lassen und mit der Standardkonfiguration aus der Anwendung heraus dann darauf zugreifen.

3.4 Anwendungsschicht

Wir wechseln in das Teilprojekt `app`. Da wir in dieser Schicht im Moment keine Funktionen implementieren, reicht es aus, wenn wir eine Konfigurationsdatei im Verzeichnis `src/main/resources/de/wicketpraxis/app/` erstellen.

Listing 3.16 app.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  <import resource="classpath:/de/wicketpraxis/ \
    persistence/persistence.xml"/>

</beans>
```

In dieser Datei wird die Konfiguration aus der Persistenzschicht eingebunden. Mehr ist an dieser Stelle nicht notwendig.

3.5 Präsentationsschicht

Wir haben jetzt alle untergeordneten Anwendungsschichten erstellt. Als letztes Teilprojekt vervollständigen wir die Präsentationsschicht.

3.5.1 Hilfsklasse für Maven-Projekte

Wicket lädt im Entwicklungsmodus die Markup-Dateien für die Komponenten bei jeder Veränderung neu. Dabei wird aber auf Dateien in `target` zugegriffen, also auf eine Kopie

von `src/main/resources`. Damit im richtigen Verzeichnis gesucht wird, müsste man Anpassungen in der Projektbeschreibung (`pom.xml`) vornehmen. Leider funktioniert diese Methode nicht zuverlässig, sodass ich nach Alternativen gesucht habe. Glücklicherweise bietet Wicket die Möglichkeit, die Anwendung in diesem Aspekt anzupassen. Die folgende Klasse sucht die Ressourcen, die Wicket benötigt, zuerst in dem in Maven-Projekten definierten Standard-Ressourcenpfad (`src/main/resources`).

Listing 3.17 MavenDevResourceStreamLocator.java

```
package de.wicketpraxis.wicket.util.resource;

...
import org.apache.wicket.util.resource.*;
import org.apache.wicket.util.resource.locator.ResourceStreamLocator;

public class MavenDevResourceStreamLocator extends ResourceStreamLocator
{
  String _prefix="src/main/resources/";

  @Override
  public IResourceStream locate(Class<?> clazz, String path)
  {
    IResourceStream located=getFileSysResourceStream(path);
    if (located != null) return located;
    return super.locate(clazz, path);
  }

  private IResourceStream getFileSysResourceStream(String path)
  {
    File f=new File(_prefix+path);
    if ((f.exists()) && (f.isFile()))
    {
      return new FileResourceStream(f);
    }
    return null;
  }
}
```

3.5.2 Wicket Web Application

Die kleinste Wicket-Anwendung besteht aus mindestens einer Seite und einer von `WebApplication` abgeleiteten Klasse. Da unsere kleine Anwendung aber z.B. bereits über eine Datenbankanbindung verfügt, wird sie an dieser Stelle etwas komplexer. Natürlich müssen wir daher auf Funktionen zurückgreifen, die bisher nicht erklärt wurden, die aber bereits einen guten Eindruck vermitteln können, wie eine Webanwendung mit Wicket funktioniert. Dieses Grundgerüst kann dann als Basis für alle folgenden Beispiele benutzt werden.

Zuerst legen wir eine Seite an, die dann in der Applikationsklasse als Startseite definiert werden kann.

Listing 3.18 Start.java

```
package de.wicketpraxis.web.pages;

...

public class Start extends WebPage
```

```
{
  @SpringBean(name=UserDao.BEAN_ID)
  UserDao _userDao;

  public Start()
  {
    LoadableDetachableModel<List<User>> userModel=
      new LoadableDetachableModel<List<User>>()
    {
      @Override
      protected List<User> load()
      {
        return _userDao.findAll(0, 10);
      }
    };

    ListView<User> userList=new ListView<User>("userList",userModel)
    {
      @Override
      protected void populateItem(ListItem<User> item)
      {
        item.add(new Label("name",item.getModelObject().getName()));
      }
    };

    add(userList);
  }
}
```

Die Annotation `@SpringBean` definiert, wie das Feld initialisiert werden soll. Dabei wird über einen optional zu definierenden Namen auf die Spring-Konfiguration zurückgegriffen und nach einer Spring-Bean vom Typ UserDao und der optionalen ID mit dem Wert aus dem Attribut `name` gesucht. Dieses Verhalten muss noch in der Applikation definiert werden.

Das Objekt `userModel` sorgt dafür, dass bei Bedarf die Liste der ersten 10 Nutzer aus der Datenbank geladen werden. Das Objekt `userList` ist für die Anzeige zuständig und fügt für jeden Eintrag der Liste ein Label mit dem Nutzernamen hinzu.

Jetzt benötigen wir noch eine passende Markup-Datei für die Seite und legen eine Datei im Verzeichnis `src/main/resources/de/wicketpraxis/web/pages/` an.

Listing 3.19 Start.html

```
<html>
  <head>
    <title>WicketPraxis</title>
  </head>
  <body>
    <h1>WicketPraxis</h1>

    <table>
      <thead>
        <tr>
          <th>Name</th>
        </tr>
      </thead>
      <tbody>
        <tr wicket:id="userList">
          <td><span wicket:id="name"></span></td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

Die Attribute `wicket:id` referenzieren dabei die Komponenten in unserer Start-Klasse. Als Nächstes müssen wir unsere eigene Applikationsklasse erstellen.

Listing 3.20 WicketPraxisApplication.java

```
package de.wicketpraxis.web;

...

public class WicketPraxisApplication extends WebApplication
{
  @Override
  protected void init()
  {
    super.init();

    addComponentInstantiationListener(new SpringComponentInjector(this));

    if (DEVELOPMENT.equalsIgnoreCase(getConfigurationType()))
    {
      getResourceSettings(). \
        setResourceStreamLocator(new MavenDevResourceStreamLocator());
    }
  }

  @Override
  public Class<? extends Page> getHomePage()
  {
    return Start.class;
  }
}
```

Beim Start der Webanwendung wird von dieser Klasse genau eine Instanz erzeugt. Dabei wird die `init`-Methode nur einmal aufgerufen. In dieser Methode werden alle Einstellungen für diese Anwendung durchgeführt. Als Erstes wird der `SpringComponentInjector` eingebunden, der für die Auswertung der `SpringBean`-Annotation sorgt. Wenn Wicket im Entwicklungsmodus ist, wird der `MavenDevResourceStreamLocator` eingebunden und lädt während der Entwicklung zuverlässig alle veränderten Markup-Dateien neu. `Start` wird als die Startseite für die Anwendung definiert.

3.5.3 Servlet-Konfiguration

Damit die Anwendung auch gestartet wird, müssen wir die Servlet-Konfiguration anpassen. Wir ändern unsere bisher leere `web.xml` im Verzeichnis `src/main/webapp/WEB-INF/`.

Hinweis

Windows unterscheidet bei Datei- und Verzeichnisnamen nicht zwischen Groß- und Kleinschreibung. Das kann dazu führen, dass es beim Bereitstellen auf einem Unix-Server zu Problemen kommen kann, wenn der Verzeichnisname nicht richtig geschrieben wurde. Der Servlet-Container erwartet die `web.xml` aber im Verzeichnis WEB-INF.

Listing 3.21 web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.4//EN"
```

```
    "http://java.sun.com/j2ee/dtds/web-app_2_4.dtd"-->

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

  <display-name>wicketpraxis.com</display-name>
  <description>
    WicketPraxis Webapp
  </description>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <filter>
    <filter-name>de.wicketpraxis.webapp</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter
    </filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>de.wicketpraxis.web.WicketPraxisApplication
      </param-value>
    </init-param>
    <init-param>
      <param-name>configuration</param-name>
      <param-value>deployment</param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>de.wicketpraxis.hibernate.osv</filter-name>
    <filter-class>
      org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>de.wicketpraxis.hibernate.osv</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>de.wicketpraxis.webapp</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

An erster Stelle wird eine Spring-Konfiguration im selben Verzeichnis angegeben (`/WEB-INF/applicationContext.xml`). Auf diese Konfiguration wird sowohl von Wicket als auch durch den `OpenSessionInViewFilter` zugegriffen. Der `ContextLoaderListener` stellt einen aus der Konfiguration ermittelten `ApplicationContext` bereit und sorgt beim Beenden des Servers für das Herunterfahren des `Context`. Im `WicketFilter` geben wir nun unsere eigene Wicket-Anwendung an und setzen den Startmodus auf `deployment`. Damit wird das Anwendungsarchiv standardmäßig mit diesem Modus erstellt. Das Öffnen und Schließen der Session für den Datenbankzugriff über Hibernate übernimmt der `OpenSessionInViewFilter`. Die Reihenfolge der Filter im Bereich `filter-mapping` ist daher

wichtig. Die Anfrage durchläuft den erstgenannten Filter vor dem nächsten. Damit ist sichergestellt, dass die Datenbankverbindung geöffnet ist, wenn die Anfrage durch Wicket behandelt wird.

3.5.4 Spring-Konfiguration

Wir erstellen jetzt im selben Verzeichnis die referenzierte Konfigurationsdatei für den `ApplicationContext`. Dabei könnten wir zwar direkt unsere `app.xml` einbinden. Ich empfehle aber einen Umweg, sodass a) jede Schicht die Konfigurationsdatei an einer nachvollziehbaren Position ablegt und b) in der Konfigurationsdatei für den Servlet-Container keine zusätzlichen Einstellungen abgelegt werden. Für den Fall, das Unit-Tests für die Präsentationsschicht durchgeführt werden müssten, wäre es sehr schwierig (wenn nicht unmöglich), an diese Datei heranzukommen.

Listing 3.22 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <import resource="classpath:/de/wicketpraxis/web/webapp.xml"/>
</beans>
```

In diesem binden wir die noch zu erstellende Datei `webapp.xml` aus dem Ressourcenpfad ein. Danach erstellen wir diese Datei im Verzeichnis `src/main/resources/de/wicketpraxis/web/`.

Listing 3.23 webapp.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  <import resource="classpath:/de/wicketpraxis/app/app.xml"/>
</beans>
```

Die Anwendung ist damit vollständig konfiguriert und kann jetzt gestartet werden.

3.5.5 Start der Anwendung

Wir haben jetzt alles zusammen. Alle Teilprojekte sollten, wenn noch nicht geschehen, über das ParentPom-Projekt erstellt (`mvn install`) werden. Dann wechseln wir in das Webapp-Projekt und starten die Webanwendung mit:

```
$ mvn jetty:run -Dwicket.configuration=development
```

Der eingebettete Webserver sollte starten und den erfolgreichen Start mit folgender Meldung quittieren:

```
...
INFO: [WicketPraxisApplication] Started Wicket version 1.4 in development mode
*******************************************************************
*** WARNING: Wicket is running in DEVELOPMENT mode.              ***
***                               ^^^^^^^^^^^                   ***
*** Do NOT deploy to your live server(s) without changing this.  ***
*** See Application#getConfigurationType() for more information. ***
*******************************************************************
2009-01-13 19:18:17.161::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Damit wurde der Server erfolgreich gestartet. Man öffnet nun mit dem Browser die Seite http://localhost:8080/. Dort erscheint eine Fehlermeldung mit einem Link, der auf unsere Webanwendung verweist. Nach einem Klick sollte man ein Ergebnis wie in Abbildung 3.1 sehen.

Abbildung 3.1 Startseite mit Daten aus der Datenbank

Wenn man wie ich schon einen Eintrag in die Datenbank getätigt hat, sieht man in der Ergebnistabelle auch schon einen Eintrag. Damit ist die Anwendung funktionsfähig, und jede der notwendigen Schichten ist erstellt und eingebunden.

Hinweis

Wicket unterscheidet zwischen den Modi Deployment und Development. Alle erweiterten Informationen, die im Development-Modus angezeigt werden, werden durch den Modus Deployment deaktiviert. Für die Entwicklung sollte der Development-Modus gewählt werden, weil das die Fehlersuche erheblich vereinfacht.

4 Die Wicket-Architektur

Nachdem wir nun unsere erste kleine Webanwendung erstellt haben, möchte ich etwas genauer auf die Architektur von Wicket eingehen, bevor wir uns in den nächsten Kapiteln mit den einzelnen Komponenten beschäftigen.

4.1 Wicket und das HTTP-Protokoll

Eine Webanwendung benutzt das HTTP-Protokoll, um mit dem Browser zu kommunizieren. Über dieses Protokoll wird eine Anfrage an den Server geschickt und die Antwort an den Browser übermittelt. Das Protokoll dient dem Transport der Daten und ist selbst zustandslos. Das bedeutet, dass eine Anfrage an den Server in keiner Beziehung zu einer vorangegangenen Antwort stehen muss.

Diese Zustandslosigkeit des Protokolls verursacht eine Reihe von Problemen, die in den verschiedenen Webframeworks unterschiedlich gelöst werden. Wicket orientiert sich dabei stark an einer Desktop-Anwendung, sodass man als Entwickler nur selten damit konfrontiert wird, dass es sich immer noch um eine Webanwendung handelt. Dennoch unterstützt Wicket den Entwickler mit einer Reihe von Schnittstellen, die weiterhin den Zugriff auf das Transportprotokoll ermöglichen oder davon abstrahiert bestimmte Funktionen zur Verfügung stellt.

4.2 Struktur

Wicket besitzt verschiedene Elemente, die in einer Webanwendung zusammenspielen. Ich werde jedes im Folgenden etwas eingehender erläutern, sodass sich ein Gesamtbild ergibt, was für das Verständnis von Wicket hilfreich ist.

4.2.1 WebApplication

Wie wir bereits gesehen haben, müssen wir für unsere eigene Anwendung eine von `WebApplication` abgeleitete Klasse erstellen. Von dieser Klasse wird auf dem Server nur eine Instanz pro Anwendung erzeugt. Die `init`-Methode wird beim Start nur einmal ausgeführt, sodass an dieser Stelle alle Einstellungen vorgenommen werden können, die für die Anwendung gelten sollen. Außerdem gibt es verschiedene Methoden, die man mit eigenen Implementierungen überladen und so die Anwendung an die eigenen Bedürfnisse anpassen kann.

4.2.2 Session

Jeder Nutzer, der auf die Anwendung zugreift bekommt eine `Session` zugewiesen. Das ist notwendig, da Wicket alle Informationen, die für eine Nutzerinteraktion notwendig sind, in dieser Session speichert. Während einige andere Frameworks versuchen, den Zustand einer Anwendung in URL-Parametern abzulegen, überträgt Wicket in dieser Phase nur die vom Nutzer durchgeführte Aktion.

Die Daten einer Session werden in einem `SessionStore` gespeichert. Normalerweise werden die Daten in einer `HttpSession` aus dem `javax.servlet`-Paket gespeichert. Man könnte allerdings auch seinen eigenen `SessionStore` implementieren.

4.2.3 PageMap

Jede `Session` hat mindestens eine `PageMap`. In dieser `PageMap` sind die Seiten (`Page`) abgelegt, die der Nutzer aufgerufen hat. Neben der zuletzt besuchten Version einer Seite finden sich in der `PageMap` auch ältere Versionen der Seite wieder. Dadurch kann der Nutzer im Browser auf die letzten Seiten zurückspringen, ohne dass es zu Fehlern kommt, weil Wicket den Zustand der Anwendung zu diesem Zeitpunkt wiederherstellen kann. Wenn ein Nutzer mehr als ein Browserfenster geöffnet hat (z.B. mit einem Popup), legt Wicket mehr als eine `PageMap` an.

Auch wenn diese Vorgehensweise dafür sorgt, dass jederzeit die korrekten Funktionen aufgerufen werden, muss man sich als Entwickler nicht mit solchen Details beschäftigen, denn dieser Vorgang geschieht automatisch und vollkommen transparent.

4.2.4 Page

Eine Seite (`Page`) ist eine Komponente, die sich von anderen Wicket-Komponenten nur dadurch unterscheidet, dass alle anderen Komponenten immer zu einer Seite gehören. Die Seite ist damit die oberste Komponente im Komponentenbaum. Man kann sich eine Page auch als Browserfenster vorstellen. Die Darstellung einer Seite liefert als Ergebnis die HTML-Daten, die der Browser dann darstellt.

4.2.5 PageStore

Wicket hält die aktuellste Seite der PageMap im Speicher vor. Alle anderen Seiten werden im PageStore abgelegt. Im Standardfall wird dazu die Seite serialisiert und in einem DiskPageStore, also auf der Festplatte abgelegt. Bei Bedarf wird die Seite wieder aus dem PageStore geladen. Auf diese Weise hält sich der Speicherverbrauch in Grenzen, ohne an Funktionalität einzubüßen.

4.2.6 Component

Eine Komponente ist die Basiseinheit einer Wicket-Anwendung. Alle Wicket-Komponenten sind von dieser Klasse abgeleitet.

4.3 Request-Behandlung

Wicket kapselt die bei einer Webanwendung beteiligten `HttpRequest`- und `HttpResponse`-Klassen. Diese werden durch einen `RequestCycle` abgearbeitet, wobei die Behandlung der verschiedenen Phasen an einen `RequestCycleProcessor` übergeben werden. Im `RequestCycle` werden folgende Phasen durchlaufen:

- `PREPARE_REQUEST`: Startet die Request-Verarbeitung.
- `RESOLVE_TARGET`: Ermittelt das Ziel dieser Abfrage (`RequestTarget`) durch den `RequestCycleProcessor`.
- `PROCESS_EVENTS`: Die Event-Behandlung durch den `RequestCycleProcessor` wird gestartet.
- `RESPOND`: Der Response wird durch den `RequestCycleProcessor` erstellt.
- `DETACH_REQUEST`: Alle temporären Daten werden gelöscht, indem für jede Komponente die Methode `detach()` aufgerufen wird. Danach wird die Seite in der PageMap abgelegt. Zusätzlich wird die Seite serialisiert, damit sie im PageStore abgelegt werden kann.
- `DONE`: Die Abarbeitung ist abgeschlossen, der nächste Request kann verarbeitet werden.

Hinweis

Wenn an dieser Stelle eine Komponente noch Referenzen auf Objekte hat, die nicht serialisiert werden können, kann Wicket die ganze Seite nicht serialisieren und somit nicht in den PageStore schreiben. Das kann zu Problemen führen, wenn man später auf diese Seite zugreifen möchte. Wenn zu einem späteren Zeitpunkt auf diese Seite zugegriffen wird, kann Wicket diese nicht im PageStore finden und gibt eine Fehlermeldung aus.

Der `RequestCycleProcessor` stellt dabei folgende Funktionen bereit:

- `resolve(RequestCycle, RequestParameters)`: Die URL und die URL-Parameter werden dekodiert und das `RequestTarget` ermittelt.

- `processEvents(RequestCycle)`: Wenn das `RequestTarget` ermittelt wurde, werden die Events verarbeitet. Dabei werden dann Events wie der Klick auf einen Link oder das Abschicken eines Formulars ausgewertet (`IRequestListener z.B. IlinkListener`).
- `respond(RequestCycle)`: Nachdem die Events verarbeitet wurden, wird das Ergebnis gerendert und an den Browser zurückgeschickt.
- `respond(RuntimeException, RequestCycle)`: Wenn ein Fehler aufgetreten ist, der nicht abgefangen wurde, wird diese Methode aufgerufen, um auf diesen Zustand geeignet zu reagieren. Im Entwicklungsmodus wird z.B. eine andere, mit mehr Informationen versehene Fehlerseite dargestellt.

Was dabei an den Browser zurückgeschickt wird, hängt natürlich davon ab, ob die Anfrage eine Seite oder ein Bild zurückliefern soll, oder ob das Ergebnis für eine Ajax-Anfrage aufbereitet werden muss.

4.3.1 Komponentenphasen

Jede Komponente hat neben dem Lebenszyklus, der mit dem Erstellen der Komponente über einen Konstruktor beginnt und mit dem Bereinigen durch den Garbage Collector endet, auch einen Request-Zyklus. Dabei sind die wesentlichen Phasen:

- Request-Behandlung: Die durch den Request beschriebene Aktion wird durchgeführt (Abschnitt 4.3.1.1).
- `onBeforeRender`: Wenn die Komponente sichtbar ist, dann wird, bevor die Komponente dargestellt wird, die `onBeforeRender`-Methode aufgerufen. Dabei kann z.B. die Sichtbarkeit manipuliert werden, was Einfluss auf diesen und den nächsten Schritt hat.
- `onRender`: wird aufgerufen, wenn die Komponente dargestellt wird.
- `onAfterRender`: wird immer aufgerufen, auch wenn die Komponente unsichtbar ist.
- `onDetach`: wird danach aufgerufen und sorgt dafür, dass die temporären Daten gelöscht werden können, damit die Session nur soviel Platz wie nötig belegt.

4.3.1.1 Request-Behandlung

Die durch den `Request` ausgelösten Aktionen werden durchgeführt. Dabei werden zuerst die Komponenten gesucht, bei der eine Aktion ausgelöst wurde. Wenn die Komponente gefunden wurde, wird die Aktion ausgelöst, die dann z.B. im Fall eines Links dazu führt, dass irgendwann `onClick()` aufgerufen wird. Bei Formularen werden die durch den `Request` übergebenen Werte der Formularkomponenten verarbeitet.

4.3.2 Nebenläufigkeit – Threads

Pro `Request` wird ein `Thread` ausgeführt, sodass man sich normalerweise keine Gedanken um dieses Thema machen muss – mit einer wichtigen Ausnahme: die Session. Da ein Nut-

zer einer Session zugewiesen ist, kann es passieren, dass der Zugriff auf die Session gleichzeitig erfolgt. Daher muss man selbst sicherstellen, dass der Zugriff synchronisiert erfolgt.

4.4 Komponenten, Modelle, Markup

Wicket ist ein MVC-Framework. MVC steht dabei für Model, View und Controller. Das Model stellt die Daten bereit, die durch den Controller verändert werden können und die durch eine View dargestellt werden.

4.4.1 Komponenten

Die kleinste Einheit einer Wicket-Anwendung ist die Komponente. Eine Komponente übernimmt dabei die Funktion eines Controllers, wobei das Framework dafür sorgt, dass die Aktion des Nutzers der richtigen Komponente zugeordnet werden kann und die Komplexität dieser Verarbeitung vollständig kapselt.

4.4.2 Modelle

Wicket orientiert sich dabei an dem Programmiermodell einer Desktop-Anwendung. In diesem Modell informiert der Controller die View darüber, ob sich etwas geändert hat und die Komponente neu gezeichnet werden muss. Da bei Webanwendungen immer (es sei denn, Ajax kommt zum Einsatz) die ganze Seite dargestellt werden muss, entfällt das Benachrichtigen der View über die Veränderung. Es werden immer alle Komponenten mit den zugehörigen Daten dargestellt. Wenn Wicket über die Modelländerungen informiert wird, zieht Wicket diese Informationen heran, um für diesen neuen Zustand eine neue Version der Seite anzulegen. Die alte Version mit alten Daten wird im PageStore abgelegt.

4.4.3 Markup

Eine Wicket-Komponente besteht aus einer Java-Klasse, die von einer Wicket-Basisklasse geerbt hat, und einer dazugehörigen Markup-Datei, die denselben Namen besitzt, in `src/main/resources` im selben Unterverzeichnis wie die Klasse abgelegt werden muss und die Endung `html` besitzt. Das bedeutet, dass eine Klasse `StartPage` im Package `de.wicketpraxis.pages` von der Klasse `WebPage` abgeleitet ist und eine Markup-Datei im Verzeichnis `de/wicketpraxis/pages/` innerhalb des Ressourcenverzeichnisses `src/main/resources` mit den Dateinamen `StartPage.html` benötigt.