

Hibernate Example

JTA, JPA, ORM, JSF



14. Juni 2016

Markus Limbeck

Inhalt

[Vorbereitungen 2](#_Toc453709355)

[Ordner Struktur 2](#_Toc453709356)

[Java Klassen 2](#_Toc453709357)

[HibernateUtil.java 2](#_Toc453709358)

[Ablauf 3](#_Toc453709359)

[User.java 3](#_Toc453709360)

[Mapping 3](#_Toc453709361)

[UserBean.java 4](#_Toc453709362)

[UserDAO.java 4](#_Toc453709363)

[Data Access Object 4](#_Toc453709364)

[Transaction 5](#_Toc453709365)

[Auslesen aus der Datenbank 5](#_Toc453709366)

[Criteria 5](#_Toc453709367)

[Hibernate.cfg.xml 6](#_Toc453709368)

[JSF 6](#_Toc453709369)

[Index.xhtml 6](#_Toc453709370)

[Output.xhtml 7](#_Toc453709371)

[CRUD-JSF Seiten 7](#_Toc453709372)

[Create.xhtml 7](#_Toc453709373)

[Read.xhtml 8](#_Toc453709374)

[Update.xhtml 8](#_Toc453709375)

[Delete.xhtml 9](#_Toc453709376)

[Web.xml 9](#_Toc453709377)

[Faces-config.xml 10](#_Toc453709378)

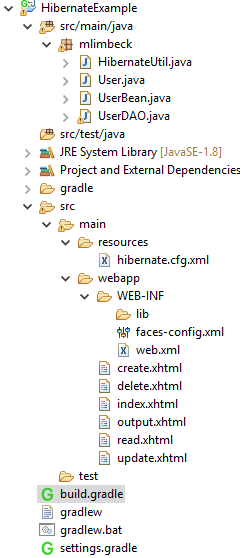
[Build.gradle 10](#_Toc453709379)

# Vorbereitungen

Prinzipiell läuft das Programm mit dem Buildtool gradle. Das heißt, solang gradle lauffähig ist, lässt sich auch das Programm ohne weiteres ausführen. Der folgende Text erklärt, wie die Anwendung in Eclipse erstellt wurde.

# Ordner Struktur

Zunächst wird eine Ordner Struktur wie folgt benötigt:



Benötigt werden 4 Java Klassen, eine Configdatei für Hibernate, build.gradle und natürlich die JSF Seiten. In den nächsten Schritten werden alle Files genauestens erklärt.

# Java Klassen

Das Programm besteht aus 4 Java Klassen. Jede Klasse hat eine genaue Aufgabe, z.B.: User.java dient für das Mapping. In den folgenden Seiten werden diese genau erklärt:

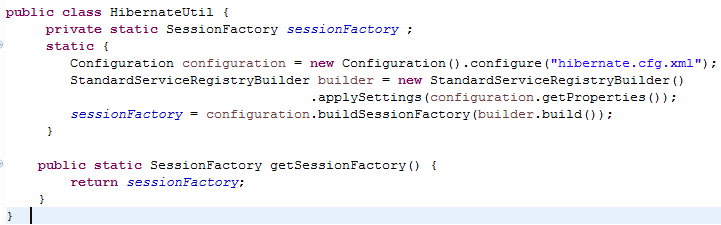
## HibernateUtil.java

Diese Klasse dient dazu eine Verbindung zu einer Datenbank aufzubauen um dann im gewünschten Zeitpunkt die Zeiten abzuspeichern.

### Ablauf

Diese Klasse erstellt eine private SessionFactory, welche über eine getMethode zugänglich gemacht wird. Für die SessionFactory wird zunächst eine Konfiguration erstellt. Die notwendigen Informationen für diese befinden sich in der „hibernate.cfg.xml“. Aber dazu später mehr.

Komplizierter ist diese Klasse auch nicht mehr.



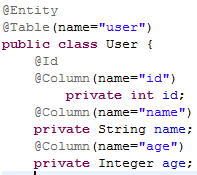
## User.java

Die User Klasse ist jene, die das Mapping übernimmt. Es werden über Annotationen die gewünschten Objekte zu den passenden Entitys gebunden.

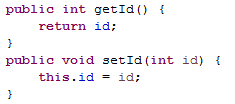
### Mapping

Zunächst muss der Klasse bekannt gegeben werden, dass es sich hier um eine Entity handelt, dies erfolgt mit der Annotation @Entity (was für ein Wunder). Weiteres wird definiert um welche Tabelle es sich in der Datenbank handelt. In diesem Fall wird die Tabelle user verwendet.

Über die Annotation @Column wird die entsprechende Spalte der Datenbank dem entsprechenden Attribute des Objektes zugewiesen. Eine kleine Sonderheit hat das Attribute id. Dieses ist nämlich in der Datenbank der Primary Key und wird über @id definiert.



Damit das Mapping auch funktioniert, müssen unbedingt den Richtlinien entsprechende Getter und Setter Methoden geschrieben werden. Hier ein kleines Beispiel wie diese Auszusehen haben:



## UserBean.java

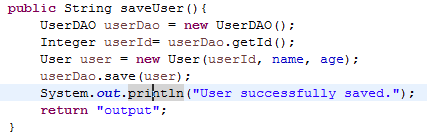
Im nächsten Schritt werden Beans benötigt. Dafür werden einfach die entsprechenden faces.beans importiert. In diesem konkreten Fall werden die Managedbean und das Requestscope benötigt.

Die benötigten Annotationen werden vor der Klasse geschrieben.



Das Herzstück dieser Klasse bietet die CRUD Methoden. Ich erkläre kurz die Create Funktion(save-Methode). Diese erstellt ein Objekt der Klasse UserDAO(Data Access Object) und dieses übernimmt dann das persistieren in die Datenbank.

Ein weiteres Detail am Rande. Die Methode wird später in einem jsf File als Action aufgerufen. Der Rückgabewert wird dann verwendet, um die Navigation zu der entsprechenden jsf Seite zu ermöglichen, welche der Browser dann anzeigen soll.



## UserDAO.java

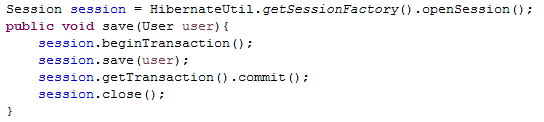
Diese Klasse führt schlussendlich die Transaction und das Persistieren der Daten durch.

### Data Access Object

Data Access Object, kurz DAO ein Entwurfsmuster, das den Zugriff auf unterschiedliche Arten von Datenquellen (z. B. Datenbanken, Dateisystem) so kapselt, dass die angesprochene Datenquelle ausgetauscht werden kann, ohne dass der aufrufende Code geändert werden muss. Dadurch soll die eigentliche Programmlogik von technischen Details der Datenspeicherung befreit werden und flexibler einsetzbar sein. DAO ist also ein Muster für die Gestaltung von Programmierschnittstellen (APIs).

### Transaction

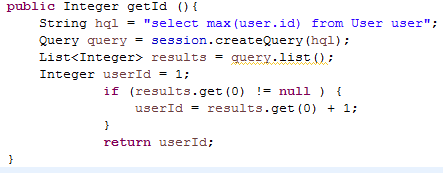
Zunächst wird eine Session erstellt. Dazu wird einfach die SessionFactory verwendet. Das Herzstück bieten wieder die CRUD Methoden. Hier wieder die Create Funktion (save Methode). Bei dieser wird die Mappingklasse User als Objekt übergeben. Die Session bietet dann eine Methode um das Objekt in der Datenbank zu speichern. Die restlichen Befehle werden nur für die Transaktion, wie breites aus dem Insy Unterreicht bekannt ist.



### Auslesen aus der Datenbank

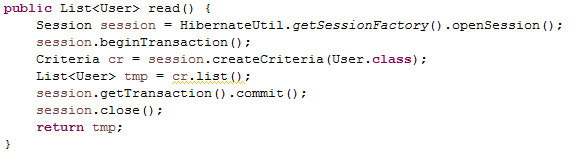
In diesem Beispiel wird die höchste id aus der Datenbank ausgelesen und um eins erhöht. Natürlich könnte dies auch über Auto Increment realisiert werden. Aber um zu demonstrieren wie aus der Datenbank ausgelesen werden kann, ist dies ganz gut.

Um aus der Datenbank Daten auszulesen, wird in diesem Fall Hibernate Query Language verwendet. Diese Sprache ist, wie nicht schwer zu erkennen, sehr ähnlich zu SQL. Der Befehl createQuery() liefert ein Objekt, welches in einer Liste alle die Spalten speichert.



### Criteria

Ein weiteres Beispiel für das Auslesen aus der Datenbank bietet die Abfragesprache Criteria. Diese ist Objektorientiert und liefert die das Ergebnis auf Objekte gemappt in einer Liste zurück. Um Citeria zu verwenden wird einfach aus der Session Criteria erstellt und erhält als Parameter die MappingKlasse( .class). Dies bietet eine wirklich einfache und gute Methode um Daten aus einer relationalen Datenbank auszulesen und direkt in Objekten zu speichern.

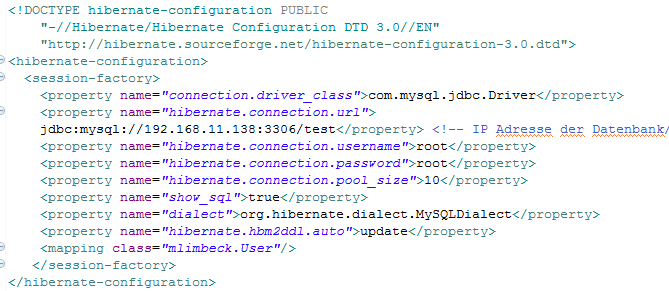


# Hibernate.cfg.xml

Dieses File speichert alle Konfigurationsinformationen für die Verbindung zur Datenbank. Dazu zählt einerseits welcher Driver verwendet werden soll, aber auch die IP Adresse und die Zugangsdaten.

Die Verbindung zu Datenbank wird in dem url property angeben. Zuerst kommt die IP-Adresse und danach der Name der Datenbank.

Wichtig ist auch, dass in der Datenbank due notwendigen Rechten vergeben werden.



# JSF

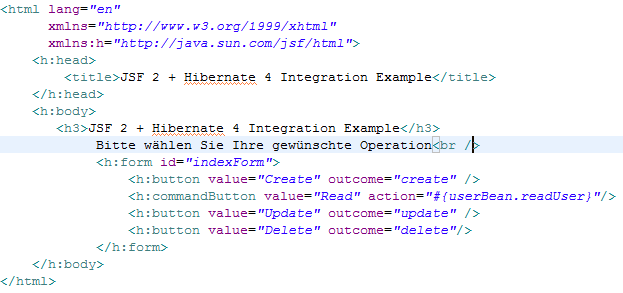
In diesem Beispiel gibt es 6 JSF Files. Eines dient nur zur Auswahl welcher der CRUD Befehle angewendet werden soll, 4 sind für die jeweiligen Befehle und die letzte Seite gibt nur eine Bestätigung, dass die Funktion erfolgreich durchgeführt worden ist.

## Index.xhtml

Die Index.xhtml ist die Startseite, welche als erstes angezeigt wird. Diese zeigt vier Buttons und lässt den User auswählen, welcher der CRUD Befehle er anwenden möchte.

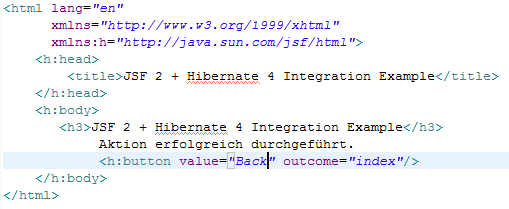
Die Navigation zu den weiteren Seiten läuft bei 3 Buttons über den Outcome(d.h.: dem Namen entsprechend wird dann die gewünschte .xhtml aufgerufen).

Der Commandbutton verwendet stattdessen den Rückgabewert (String) der Funktion, welche die Managedbean aufruft.



## Output.xhtml

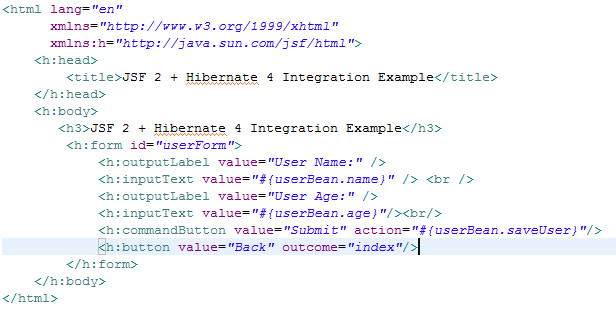
Die Output.xhtml wird aufgerufen nachdem einer der CRUD Befehle ausgeführt wird. Sie gibt einfach nur dem User Bescheid, dass seine Aktion erfolgreich durchgeführt worden ist. Weiteres besitzt sie einen Button, welche über den outcome eine Navigation zur index.xhtml.



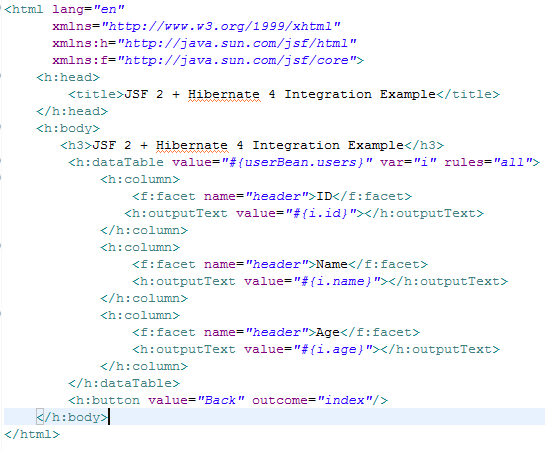
## CRUD-JSF Seiten

Die JSF Seiten sind selbst erklärend, daher wird nur ein Bild hineinkopiert.

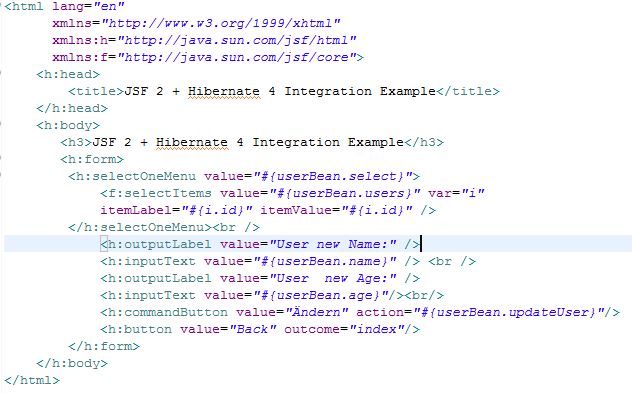
### Create.xhtml



### Read.xhtml



### Update.xhtml



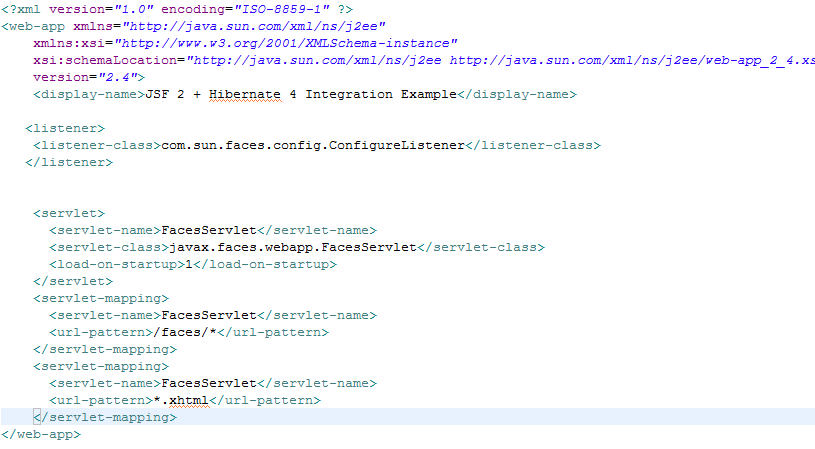
### Delete.xhtml



# Web.xml

Bei der Web.xml hat sich nichts geändert von den vorherigen Projekten.

Web.xml für JSF:



# Faces-config.xml

In dem faces-config.xml werden die notwenigen ManagedBeans definiert. Jedes ManagedBean benötigt einen Namen, eine dazugehörige Klasse und zuletzt noch ein Scope.



# Build.gradle

Auch hier sind nicht wirkliche Neuerungen vorhanden. Zusätzlich wird noch bei den Dependencies die nötige Source für Hibernate eingebunden.

