Spring and Hibernate

Ken Kousen
Will Provost

Version 2.0_3.2

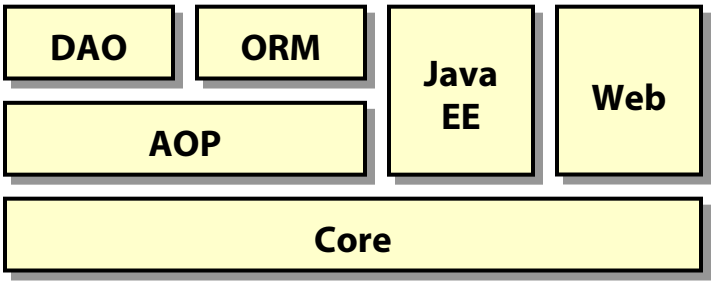
A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

Introduction

- **This presentation illustrates the use of two popular frameworks in concert:**
  - **Spring** – version 2.0
  - **Hibernate** – version 3.2
- **Though it may be interesting on its own (to those with experience in one or more of the above frameworks), we've built it primarily to compliment Capstone courses on these subjects.**
  - It will be most effective as a sort of encore presentation at the end of a class that uses one or both of those courses.
  - It uses a build-and-deploy infrastructure similar to that found in the courses – merging some of the paths to JAR files is most of it.
- **We discuss each of these frameworks very briefly before addressing the issues involved in integrating them for Jara persistence tiers.**

Spring

- The **Spring Framework** proposes to simplify and improve J2EE development, in a few key ways:



  - Its Core module provides a **lightweight container** for ordinary JavaBeans implements **inversion of control** (**IoC**) and **dependency injection**, allowing complex graphs of objects to be declared rather than managed programmatically.
  - It **simplifies exception handling** by a combination of strategies: minimizing the use of checked exceptions, and providing an excellent error reporting framework.
  - It offers a flexible **web framework** that implements the **model/view/controller pattern** and offers **pluggable strategies** for all of its high-level decision-making.
  - It implements **aspect-oriented programming** at many levels, allowing "cross-cutting concerns" to be addressed effectively for even very complex applications.
  - It offers a simple means of **integrating persistence tools** while preserving application-level control of transactions.

Hibernate

- **Hibernate is an open source project whose purpose is to make it easy to integrate relational data into Java programs.**
- **This is done through the use of XML mapping files, which associate Java classes with database tables.**
- **Hibernate provides basic mapping capabilities. It also includes several other object/relational mapping (ORM) capabilities, including:**
  - An enhanced, object-based SQL variant for retrieving data, known as **Hibernate Query Language** (**HQL**).
  - Automated processes to **synchronize** objects with their database equivalents.
  - Built-in database **connection pooling**, including three open-source variants.
  - **Transactional capabilities** that can work both stand-alone or with existing Java Transaction API (JTA) implementations.
- **The goal of Hibernate is to allow object-oriented developers to incorporate persistence into their programs with a minimum of effort.**

Spring and Hibernate

- **Despite their conflicting seasonal allusions, these two frameworks can cooperate neatly within the persistence tier.**
- **Hibernate can handle persistence more or less by itself.**
  - It offers **object/relational mapping**.
  - It can manage **sessions** and **transactions**.
- **Spring brings a few nice features not found in Hibernate:**
  - The **IoC container** makes configuring data sources, transaction managers, and DAOs easy.
  - It manages the Hibernate **SessionFactory** as a **singleton** – a small but surprisingly annoying task that must be implemented manually when using Hibernate alone.
  - It offers a **transaction** system of its own, which is **aspect-oriented** and thus configurable, either through **Spring AOP** or **Java-5 annotations**. Either of these are generally much easier than working with Hibernate's transaction API.
- **As good as Hibernate is, Spring makes it a bit better.**
  - You write less code.
  - Transaction management becomes nearly invisible for many applications, and where it's visible, it's still pretty easy.
  - You integrate more easily with other standards and frameworks.

Configuring Hibernate Using Spring

- **A typical Hibernate application configures its SessionFactory using a properties file or an XML file.**
- **First, we start treating that session factory as a Spring bean.**
  - We declare it as a Spring **<bean>** and instantiate it using a Spring **ApplicationContext**.
  - We configure it using Spring **<property>**s, and this removes the need for a **hibernate.cfg.xml** or **hibernate.properties** file.
  - Spring **dependency injection** – and possibly **autowiring** – make short work of this sort of configuration task.
- **Hibernate object/relational mapping files are included as usual.**
- **So we might write:**

```
<bean
  id="sessionFactory"
  class="org.hibernate.impl.SessionFactoryImpl"
>
  <property
    name="connection.driver_class"
    value="com.vendor.Driver"
  />
  <property name="connection.url" value="..." />
  <property name="show_sql" value="true" />
  ...
</bean>
```

  - ... but we're about to see that Spring offers a session factory for Hibernate that enables it to take advantage of AOP transactions.
  - So the code above is hypothetical: just a step in the right direction.

Letting Spring Manage Transactions

- **To integrate Spring's aspect-oriented transaction control with Hibernate's ORM, Spring provides a special session factory, LocalSessionFactoryBean.**
  - This and the rest of Spring's Hibernate 3.x support are found in package **org.springframework.orm.hibernate3**.
  - This factory will create and manage proper Hibernate **Session**s, and also knows about Spring transactions.
  - It is configured through a set of properties that are closely related to the properties found in a Hibernate configuration file, though some names and data types are different.
- **Declare an instance of HibernateTransactionManager and inject the session factory into it.**
- **Then define transactional advice on relevant persistent classes.**
  - There are several ways to do this; we focus on the use of Java-5 annotations, meaning that the configuration will include the element **`<tx:annotation-driven>`**.
  - Then individual persistent classes or (most likely) related DAOs will announce that they or some of their methods are **@Transactional**.
- **It's essential that all these objects be instantiated through a Spring application context as managed beans.**
  - Otherwise, there will be no AOP proxies and hence no "hook" for transaction control; everything else might work, but transactions would not be in force.

Tools and Environment

- **The first code example runs on the following technology stack:**
  - **Java 5.0**
  - **MySQL 5.0**
  - **Hibernate 3.2**
- **The second one is a web application and so brings in a few more players:**
  - **Tomcat 5.5**
  - **JSTL 1.1**
  - **Spring 2.0**
- **Everything but the 5.0 JDK is bundled with the examples, in directories under c:/Capstone/Tools.**
- **Be sure the environment includes these settings:**
  - An environment variable **CC_MODULE** must be set to **c:\Capstone\Spring+Hibernate**.
  - The executable path must include the JDK's **bin** directory and also **c:/Capstone/Tools/Ant1.6/bin**.

MySQL and Tomcat

- **Start the MySQL RDBMS now, by running the following command from c:/Capstone/Tools/MySQL5.0/bin:**

```
mysqld
```

  - Note that the console you use to run this command will appear to hang; this is the MySQL daemon process listening for connections at its default port 3306.
  - When you want to shut the server down again, run this prepared script from the same directory:

```
shutdown
```

- **Start the Tomcat server as well, by running the following script from c:/Capstone/Tools/Tomcat5.5/bin:**

```
startup
```

  - You'll see a new console appear for the newly-started server process, and after a few seconds Tomcat will be initialized, showing a confirmation like this:

```
INFO: Server startup in 6229 ms
```

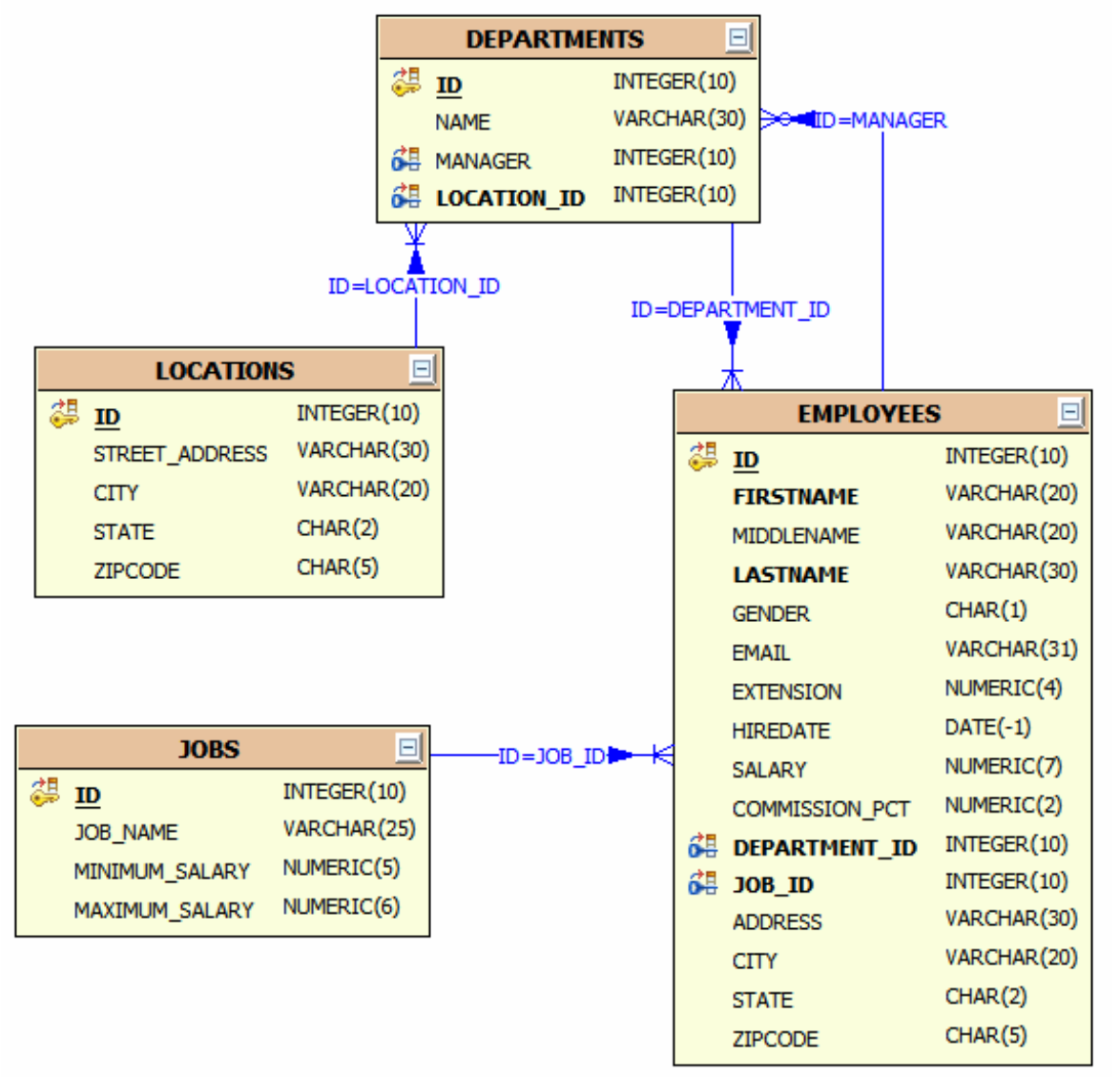
  - When you want to shut down Tomcat, run this script from the same directory:

```
shutdown
```

Spring for Hibernate DAOs

EXAMPLE

- In **c:/Capstone/Spring+Hibernate/Examples/Earthlings**, the human-resources DAOs have been refit to work with Spring.
- The ERD for the HR database is shown below:



  - See **createdb.sql** for the full DDL and DML that sets up this database for MySQL.

Spring for Hibernate DAOs

EXAMPLE

- See the Spring configuration file **src/DataSourceBeans.xml**:
  - We declare a MySQL **DataSource**:

```
<bean
  id="myDataSource" class=
    "com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
>
  <property name="url" value=
    "jdbc:mysql://localhost/earthlings" />
  <property name="user" value="earthlings" />
  <property name="password" value="earthlings" />
</bean>
```

  - The session factory depends on that and also on a set of Hibernate mapping files:

```
<bean id="mySessionFactory" class=
  "org.springframework.orm.hibernate3
    .LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource" />
  <property name="mappingResources">
    <list>
      <value>cc/db/beans/Department.hbm.xml</value>
      <value>cc/db/beans/Location.hbm.xml</value>
      <value>cc/db/beans/Employee.hbm.xml</value>
      <value>cc/db/beans/Job.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
      hibernate.dialect=
        org.hibernate.dialect.MySQLInnoDBDialect
      hibernate.current_session_context_class=
        thread
    </value>
  </property>
</bean>
```

Spring for Hibernate DAOs

EXAMPLE

– We announce that our transactions are based on Java-5 annotations, and establish a transaction manager:

```
<tx:annotation-driven
  transaction-manager="myTxManager"/>

<bean id="myTxManager" class=
  "org.springframework.orm.hibernate3
    .HibernateTransactionManager">
  <property name="sessionFactory"
    ref="mySessionFactory"/>
</bean>
```

– Finally, we declare individual DAOs:

```
<bean id="empDAO" class="cc.db.dao.EmployeeDAO">
  <property name="factory" ref="mySessionFactory" />
</bean>
<bean id="deptDAO" class="cc.db.dao.DepartmentDAO">
  <property name="factory" ref="mySessionFactory" />
</bean>
<bean id="locDAO" class="cc.db.dao.LocationDAO">
  <property name="factory" ref="mySessionFactory" />
</bean>
<bean id="jobDAO" class="cc.db.dao.JobDAO">
  <property name="factory" ref="mySessionFactory" />
</bean>
<bean
  id="criteriaQueries"
  class="cc.db.hibernate.CriteriaQueries"
  autowire="byType"
/>
```

Spring for Hibernate DAOs

EXAMPLE

- With this configuration we have almost everything we need.
- But, for this to work, we have to advise Spring's transaction manager as to which methods should be transactional.
  - See **src/cc/db/dao/EmployeeDao.java** as an example of what all the DAO classes do:

```
@Transactional
public class EmployeeDAO {
  private SessionFactory factory;

  public SessionFactory getFactory() {
    return factory;
  }

  public void setFactory(SessionFactory factory) {
    this.factory = factory;
  }

  @SuppressWarnings("unchecked")
  public List<Employee> findAll() {
    return factory.getCurrentSession()
      .createCriteria(Employee.class).list();
  }
  ...
```

  - The **@Transactional** annotation is recognized by the Spring transaction manager, and the session factory creates a transactional session for the duration of each method call.

Spring for Hibernate DAOs

EXAMPLE

- Various console-application classes test the DAOs and Hibernate mappings.
- The general modus operandi is to get an instance of a DAO from the Spring application context (which will be wired to all the session and transaction goodies), and start making method calls.
  - See **src/cc/db/hibernate/PrintEmployees.java**:

```
public static void main(String[] args) {
  ApplicationContext context =
    new ClassPathXmlApplicationContext
      ("DataSourceBeans.xml");

  EmployeeDAO eDAO =
    (EmployeeDAO) context.getBean("empDAO");

  List<Employee> emps = eDAO.findAll();
  for (Employee e : emps) {
    System.out.println(e.getLastName() +
      " has job " + e.getJob().getName());
  }
}
```

  - **src/cc/db/hibernate/CriteriaQueries.java** works a little differently, instantiating itself as a sort of free-agent DAO that runs various queries that span entities, using a Hibernate **Session** directly.
  - For this to work, it too must be **@Transactional**, and so derives the current session from the session factory prior to carrying out Criteria API queries on that session.

Spring for Hibernate DAOs

EXAMPLE

- **Build and test the example as follows:**
  - Create the database:

```
createdb
```

  - Build the application:

```
ant
```

  - Test various applications:

```
run cc.db.hibernate.PrintEmployees
Acosta has job Manager
Amdell has job Junior Engineer
...
Young has job Tester
Zimmerman has job President

run cc.db.hibernate.PrintJobs
run cc.db.hibernate.PrintDepartments
run cc.db.hibernate.PrintLocations
run cc.db.hibernate.CriteriaQueries
Conjunction:
Walker Calhoun $38,000.00 MA
Hugh Campbell $32,000.00 MA
King Cardenas $84,000.00 NC
Mariana Castillo $67,000.00 NC
Total: 4
...
Number of employees in each department
[32, GA]
[22, MA]
[46, NC]
[35, NJ]
```

- **Full results can be seen in ConsoleLog.txt.**

Using Hibernate in Spring Web Applications

- **It's fairly simple to use Hibernate in a Java web application.**
  - Hibernate session factories find configuration and mapping files on the **class path**, which makes it easy to assure that these resources will be available at request time.
  - If **data sources** must be found by **JNDI**, there's a little more work to do ... but then we're going to assume Spring is in play.
  - The upcoming example uses a data-source bean configured directly for a Spring container, but Spring's **Java EE configuration schema** provides a simple means of exposing a JNDI-named data source as a bean, if necessary.
- **A "Spring MVC" web application declares its request-handling components – handler mapping, controllers, view resolver, views, and more – as Spring beans.**
- **One or more of these beans – say, a given controller that's responsible for carrying out some work on the database as its way of responding to an HTTP request – might depend on an injected Hibernate DAO.**
- **From there, everything we've already seen obtains equally in the web-server process for the web application.**
  - The DAO can be marked **@Transactional**.
  - The Spring/Hibernate session factory, once found by the web component(s), will take it from there.

Hibernate in Spring MVC

EXAMPLE

- In **c:/Capstone/Spring+Hibernate/Wholesale** is a web application that uses a Hibernate DAO to manage a single-table database of sales-feed information.
- There are two configuration files.
- **docroot/WEB-INF/Wholesale-servlet.xml** configures the web application itself.
- There's a lot here, but the connection to Hibernate is almost invisible, because ...
  - Various beans connect to the DAO by autowiring.
  - JSPs use a custom tag to look up the DAO for themselves.
- But notice that this file imports **docroot/WEB-INF/Database.xml**.
  - This is almost identical in form to the configuration file in the previous example: data source, session factory, transaction manager, and DAO.
  - It uses autowiring where the previous example didn't.
  - It refers to just a single Hibernate mapping file, **Order.hbm.xml**.

Hibernate in Spring MVC

EXAMPLE

- The DAO class **cc.sales.db.OrderDAOImplHibernate** implements the interface **cc.sales.OrderDAO**.
- This interface marks all three of its methods **@Transactional**, and so the implementing classes don't need to.
- Here's the **save** method from this DAO:

```
public void save
  (List<Order> feed, String feedName)
{
  Session session = getSession ();
  session.createQuery ("delete from Order ord " +
                       "where ord.feed = :feedName")
      .setParameter ("feedName", feedName)
      .executeUpdate ();
  for (Order order : feed)
  {
    order.setFeed (feedName);
    session.save (order);
  }
}
```

  - Note that this method leaves the data in an inconsistent state while in-process: it deletes all records for a given feed, and then reconstructs them.
  - Transactionality will be very important here.

EXAMPLE

- **Build and test this application as follows:**
  - Build the database:

```
ant create-DB
```

  - Build and run a console application that primes the table with data:

```
build
run
```

  - Build and deploy the application:

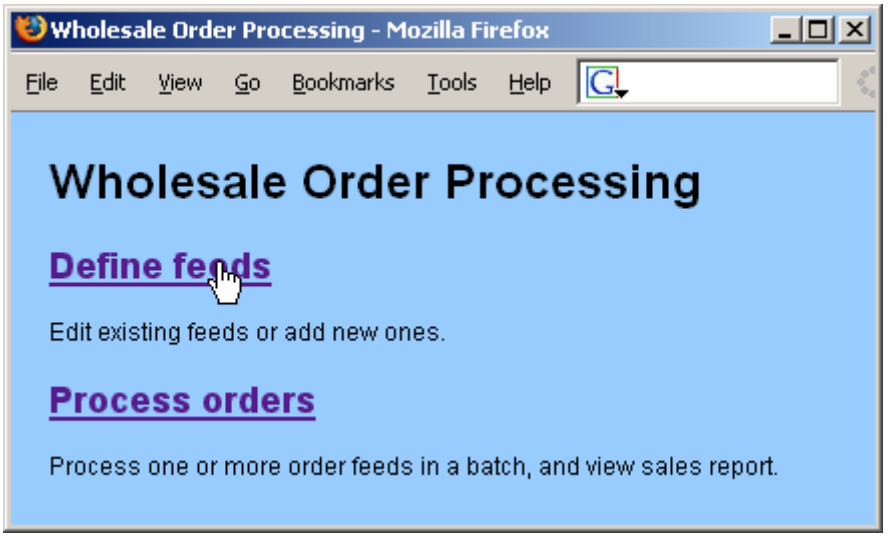
```
ant
```

  - Within 30 seconds you'll see Tomcat detect the new context and install the application:

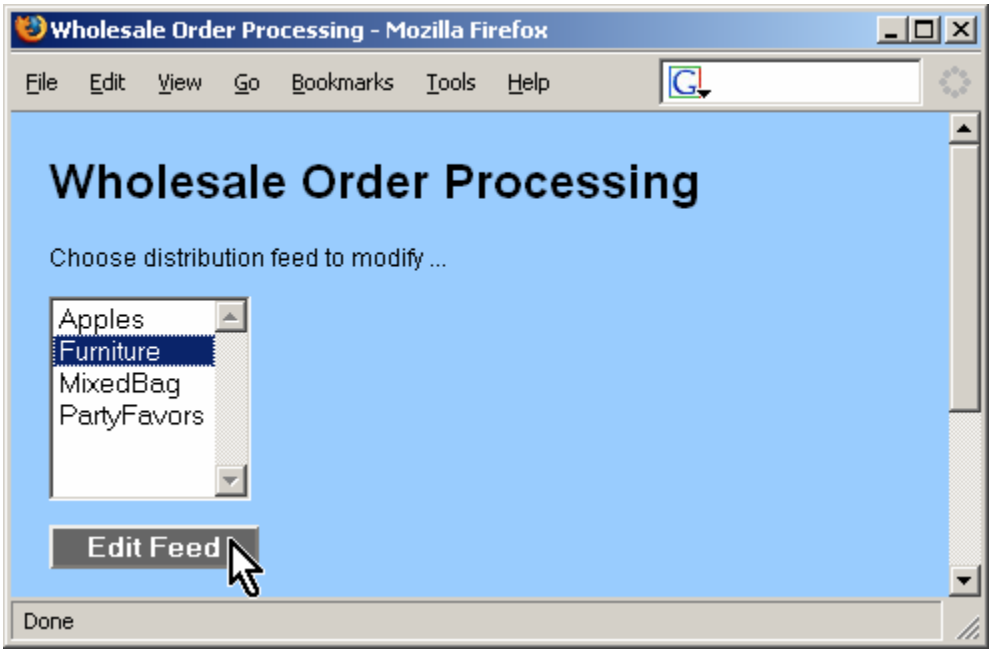
```
INFO: Servlet 'Wholesale' configured successfully.
```

– Visit the application at the following URL:

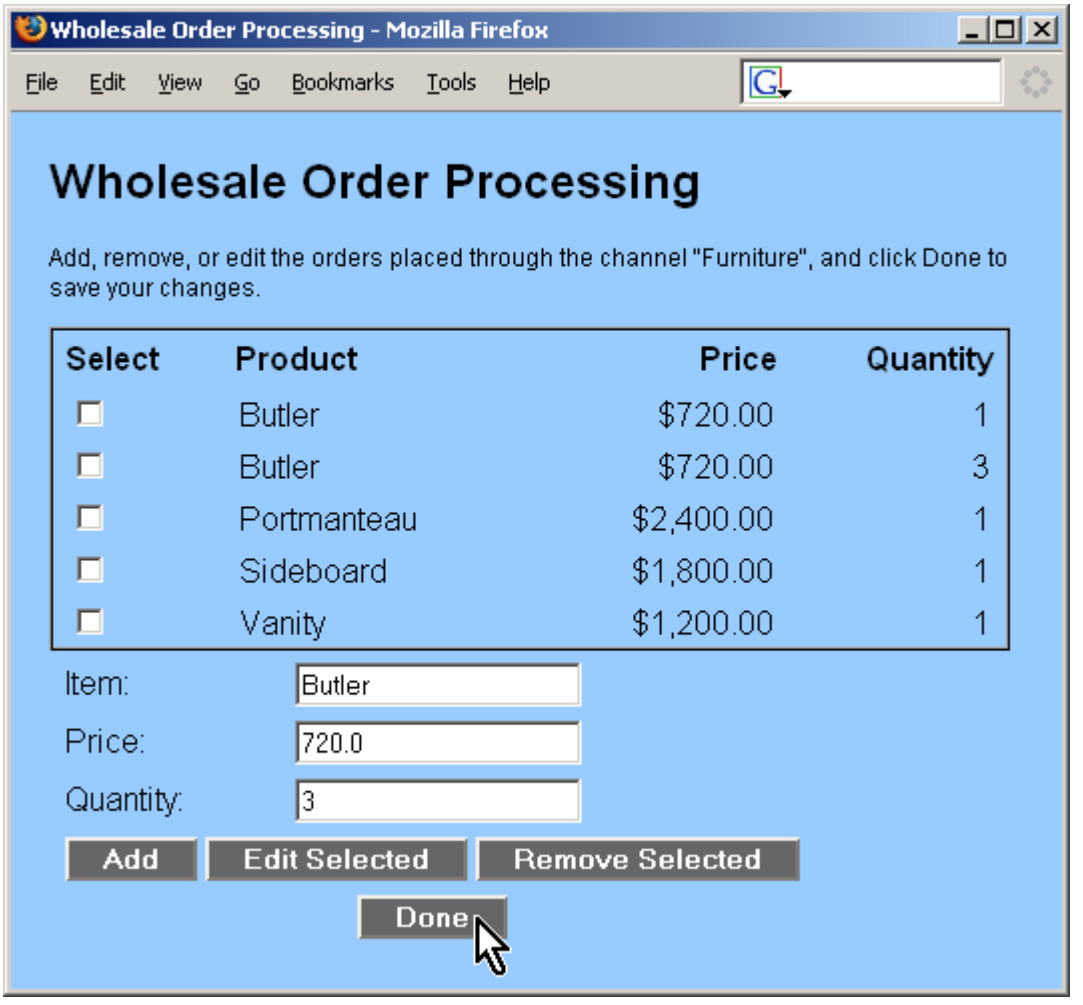
`http://localhost:8080/Wholesale`



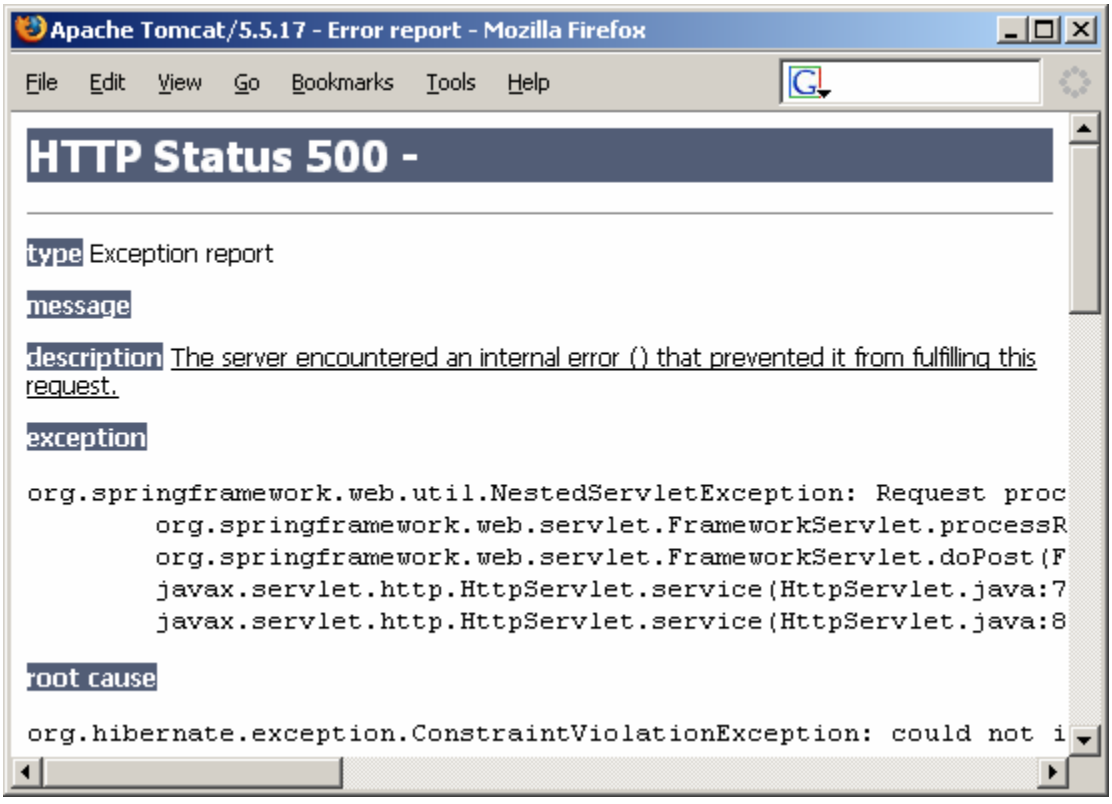
– Click on **Define feeds** and choose one of the existing feeds to edit:



– Choose the second order in the list, edit it so that it's actually just more of the product shown in the first order – as shown below. This will violate a unique-key constraint in the database that says we can't have more than one order for the same product in the same feed.



- Click **Edit Selected**, and then click **Done**.



- **The fact that we get an exception here is not a failure of Spring or Hibernate (except perhaps in the sense that we could have some more user-friendly error reporting).**
- **What we want to see is, did the data get corrupted when the user committed this error?**
  - Visit the home page again, and navigate to the same feed you just edited.
  - You will see that it is safe and sound: the entire **save** operation was rolled back, thanks to a transaction that was put in play by Spring's transaction manager for the Hibernate session used by the method.

One Little Problem ...

- **Hibernate relies on a tool called CGLIB to generate proxy classes that monitor calls to persistent objects and thus serve as Hibernates "eyes and ears."**
- **CGLIB generates classes into an area known as the "PermGen" memory space.**
- **Hibernate doesn't explicitly clean up these generated proxies, and the PermGen space isn't garbage-collected (because the Java VM assumes that it's only for class information, which should be stable).**
- **The result is a significant memory leak for certain uses of Hibernate.**
- **Standalone applications rarely run into any trouble.**
- **Web applications are another story: for Tomcat at least, repeated redeployment of a Hibernate-enabled web application will gradually drain the PermGen memory and halt the server!**
  - This won't be a problem for production environments with dedicated servers such that the server is stopped and restarted on a redeployment.
  - It is a major problem for developers who want to redeploy frequently, and for shared-server production environments.
  - There are many suggested fixes to this odd bug, but in our research we have yet to see any of them work.