<https://habr.com/post/178865/>

Разработка веб приложения на основе Java EE+JSP+Hibernate+Maven+Spring MVC в NetBeans. Часть 1



# Разработка веб приложения на основе Java EE+JSP+Hibernate+Maven+Spring MVC в NetBeans. Часть 1

* [Разработка веб-сайтов](https://habr.com/hub/webdev/),
* [Java](https://habr.com/hub/java/)
* [Из песочницы](https://habr.com/sandbox/)

#### Вступление

Это будет серия статей, помогающих начинающим Java разработчикам в их нелегком пути. Покажу пример создания интернет магазина.  
Первое что вам понадобится — это NetBeans 7.3. Скачать можно [здесь](https://netbeans.org/downloads/).   
Почему именно он? Думаю для начинающих в ЕЕ — это идеальный вариант.   
  
Используемые технологии:  
  
1) NetBeans 7.3;  
2) Maven;  
3) Hibernate;  
4) Spring MVC;  
5) JSP+JSTL;  
6) Знание про Сервлеты, Сессии, JavaBean, XML и т.д.;  
7) HTML+CSS (немного красоты по Вашему вкусу, но лучше заплатить левым людям, — у Вас с серверной стороной итак хлопот хватит);  
8) Java SE (знание коллекций, умение обрабатывать исключения… В общем, стандартный набор);  
9) Знание паттернов проектирования (DAO, Factory);  
10) JPA;  
11) SVN;  
12) SQL (для написания скриптов, заполняющих наши БД).  
  
Начнем с того, где мы будем хранить наш код и как обмениваться им с друзьями. Думаю, не все начинающие знают про репозитории, и эта часть именно для них. 

#### SVN

Есть такое понятие, как репозиторий — удаленный сервер хранения кода.  
Если Вам дадут тестовое задание, и Вы пошлете его архивом, то Вас скорее всего тоже пошлют. Может, не пошлют, но точно разочаруются в Вас.  
Существуют различные репозитории CVS, SVN, Git. Для начинающих я считаю оптимальным SVN. Ну а Вам главное не знать что это, а уметь применять. Пока достаточно будет и этого, остальное придет с опытом.   
  
Итак, есть сайт [www.assembla.com/](http://www.assembla.com/) — Вам нужно там зарегистрироваться. После регистрации нужно создать на этом сайте открытый репозиторий на стартовой странице. После регистрации будет . Cоздать свое собственное пространство.   
  
Далее публичный проект, и первый будет Subversion (SVN). Даете ему имя и заканчиваете создание. После этого вверху, на черной полоске сайта, отобразится его имя, Вы нажимаете по нему — и вас перенаправит на страницу Вашего пространства. Там будет закладка «Исходный код/SVN». В этой вкладке будет ссылка на Ваш проект. Если ничего не получится, попробуйте найти видео как это делать на [www.youtube.com/](http://www.youtube.com/).   
  
После всего этого у Вас будет ссылка на Ваше пространство. Чтобы понять, что это такое — нужно взять один из ваших проектов (или создайте какой нибудь пустой проект в NetBeans). Нажимаете на нем правой кнопочкой и у Вас в меню будет доступно «Управление версиями» -> «импортировать в репозиторий Subversion». После этого будет диалоговое окно, в котом будет путь к репозиторию — это ссылка которую Вы получили на сайте во вкладке «Исходный код».   
  
Далее, полностью удалите проект, который вы закоммитили. Дальше зайдите в папку, где у вас проекты хранятся, и проверьте что реально все удалено. Потом возвращаетесь в NetBeans и ищете в панели меню вкладку Группа(на панели где Файл, Вид, Правка, Переход, Источник...) в нем есть наш Subversion. И в подменю на нем есть «Получить». Дальше в диалоговых окнах надо будет указать ссылку на репозиторий (это ссылка которую вы получили на сайте во вкладке «Исходный код».) И когда он предложит выкачивать папки, то по дереву репозитория нужно будет найти ваш проект и выбрать его, и по окончанию вы выкачаете свой проект. Вот так происходит обмен кодом.   
Ваш проект будет постоянно синхронизироваться с репозиторием и помечать файлы, которые были изменены, или новые(то что отличается от версии на репозитории). Чтобы обновить, нужно вызвать контекстное меню, и в закладке «Управление Версиями» будет большой список того, что можно делать с проектом. «Обновлять» — это обновить свои файлы; «Фиксировать» — ложить код который Вы написали или изменили в репозиторий; «Сбрасывать» — возвращаться к версии на репозитории, и «Сравнивать» — отображение изменений строк которые отличаются от удаленных. Это способ командного обмена кодом, который используется всегда и нужно к нему привыкать. 

#### Начало

Итак, если нам нужно с кем-то поделиться кодом, показать его кому то,- мы теперь вооружены хорошеньким инструментом для этого.  
  
Вы уже скачали NetBeans, поигрались с SVN — теперь перейдем к делу. Создаете проект. Нажимаете «Создать проект», там выбираете Maven-> Веб-приложение. Называете как хотите, это все Ваша фантазия. Итак, у нас есть веб-приложение, сборка нашего проекта идет мавеном, у нас есть цель и теперь наступило время подумать над тем, как ее осуществить. То есть Вы, как разработчик, должны подумать над тем, как будет выглядеть Ваше приложение, какую иметь архитектуру, дерево пакетов и так далее. Общее количество строк кода здесь около 4000 и лучше позаботиться о красивой архитектуре заранее, иначе потом Вы просто не будете понимать что где и как у Вас работает, каким чудом Вы, к примеру, выводите последнюю купленную вещь, как считаете общую сумму покупок. И если Вас потом попросят что-то доделать или добавить — Вы осознаете что проще написать все с нуля.   
  
Ну и конечно нам нужно прикинуть наш план действий. 

##### **Итак: План действий**

1) описываем entity(entities) — сущности. Это POJO — класс, связанный с БД с помощью аннотации (@Entity) или через XML. Использовать будем JPA, поэтому импортировать надо javax.persistence.\* Почему не Hibernate Persistence API, т.к если использовать его и потом мы захотим сменить ORM библиотеку, то придется переписывать и наши классы, а JPA — это стандарт от Sun. По поводу того, что такое JPA — ну для Вас могу своими словами сказать так: это библиотека предоставляющая API для работы с \*долго живущими\* объектами, то есть позволяет нам удобно сохранять наши объекты в БД. Могу дать совет: создайте для этого отдельный пакет назовите его entity или domain — как хотите, главное чтобы Вам было понятно. В конечном итоге это может выглядеть так: edu.shop.model.domain.Customer.java edu.shop.model.domain.Notebook.java.   
Подробнее буду описывать непосредственно при рассмотрении данного пункта. Сейчас задача стоит прикинуть план действий.  
  
2) Создаем HibernateUtil (вообще суффикс или приставка Util подразумевает, что код в этом классе есть универсальный и используется множеством классов).  
Итак, в HibernateUtil мы размещаем SessionFactory. Он тяжеловесный. Этот код, по идее, должен быть независим от всего приложения, так как он устанавливает соединение с базой данных при старте и должен нам давать только Сессии с базой данных. Еще мы в этом классе регистрируем наши классы-сущности. Подробнее про этот класс расскажу позже. Засунем его тоже в отдельный пакет, к примеру, edu.shop.model.hbutil.HibernateUtil.java  
  
3) Пишем DAO.   
Что в нем писать? Пишем то, что мы хотим получить от базы данных, но нам не нужно думать как получились эти данные, важен результат. К примеру, мы определяем интерфейс ProductDAO и пишем в нем методы  
List<AnyEntity<>> getAllProducts(); потом пишем его реализацию ProductDAOImpl.  
  
В чем идея? Если бы это приложение писал я и Вы, Вы бы сказали: «Миха, мне нужны от БД следующие данные: все товары что у меня есть в БД». Я отвечаю: «не вопрос». И далее следующее развитие событий: вы в своем коде, везде где нужно делать запросы к базе пишете следующее%  
  
\*здесь обращение к методу\*.getAllProducts(); — и видите, что компилятор не ругается, а реализацию этого интерфейса я мог еще не успеть написать. И какой итог? У Вас все скомпилилось, а рабочего кода даже нет. Здесь мы применим Enums и паттерн Factory, и еще кое-что, но всему свое время. Именно в DAO нужно уделить особое внимание обработке исключений, хотя бы генерировать страницы с ошибками. Чтобы Вы быстро находили кусок неработающего кода. Иначе, Вы просто замучаетесь с отладкой.   
  
3)Здесь начнется наша работа с Spring MVC. Это долгая история и этому будет посвящена отдельная статья. Сразу скажу — это самое сложное в этом приложении. Но я Вам покажу и более простой вариант, как выводить все, не особо заботясь про паттерн MVC.   
Затронем использование скриплетов.  
  
4) Здесь у нас будут вспомогательные классы, добавляющие всякие вкусности в наш проект: подсчет общей суммы покупок; последняя купленная вещь; вспомогательные переменные, которые пригодятся нам для работы метода, который, к примеру, будет выводить нам с БД вещи не дороже 5000 грн, или не дешевле 500; вывод всех ноутбуков марки Асус. Этот пункт тесно взаимосвязан с предыдущим.  
  
Пока остановимся на этом. С остальным разберемся чуть позже. Итак, у нас есть намеченный план, приступим к его реализации.

### Entity

Мы создали наш проект и создали наш пакет с сущностями. Пусть это будет edu.shop.entity. Там мы создаем такие классы:  
  
Примечание. Корзина, в которой будут храниться наши купленные товары, будет как таблица в БД, это я сделал т.к на ней покажу некоторые основы работы с БД. Для реального случая целесообразнее будет использовать коллекции для хранения наших товаров.   
  
1) Product  
2) Notebook  
3) Camera  
4) Book  
5) Cable  
6) Customer  
7) Cart  
  
Поговорим немного о том, что такое класс-сущность.  
Entity (Сущность) — POJO-класс, связанный с БД с помощью аннотации (@Entity) или через XML. К такому классу предъявляются следующие требования:  
  
— Должен иметь пустой конструктор (public или protected);  
— Не может быть вложенным, интерфейсом или enum;  
— Не может быть final и не может содержать final-полей/свойств;  
— Должен содержать хотя бы одно @Id-поле.  
  
При этом entity может:  
  
— Содержать непустые конструкторы;  
— Наследоваться и быть наследованным;  
— Содержать другие методы и реализовывать интерфейсы.  
  
Entities могут быть связаны друг с другом: один-к-одному, один-ко-многим, многие-к-одному и многие-ко-многим.  
  
Использовать мы будем аннотации. И тут сразу у нас появляется возможность описать двумя способами. Либо мы будем писать аннотации непосредственно над полями, либо над геттерами. Скажу одно: правильно писать над геттерами, а причину Вы спросите в гугле. Не могу я все темы абсолютно здесь описать.  
  
Есть еще одно что хочу сказать. Для этого придется показать 2 примера описания класса сущности. Итак, первый пример:  
Коментарии к нему я написал в самом коде:

import java.io.Serializable;

import javax.persistence.\*;

import javax.validation.constraints.Size;

*/\*\**

*\**

*\* @author Mikhail Shumenko*

*\*/*

@Entity *//Этой аннотацией мы указываем, что данный класс является сущностью.*

@Table(name = "CART")*// Этой аннотацией мы указываем, что за эту сущность в БД отвечает таблица с именем CART*

*//Хочу отметить, что регистр не важен, эту анотацию можно не указывать, тогда хибернейт создаст нам БД с*

*//именем как у класса*

public class CartEntity implements Serializable {

*//Здесь мы пишем аннотации над полями. Правильно писать над геттерами*

*//Описываем Id таблицы*

@Id

*//Указываем, что это поле класса отвечает за столбец в таблице с именем Id*

*//Если мы его указывать не будем, хибернейт создаст столбец с именем как у поля.*

@Column(name = "ID")

*//Здесь написать можно много)) Почему я написал здесь так? В общем можно в*

*//@GeneratedValue(strategy=GenerationType.вместо TABLE написать AUTO) тогда*

*//при первой загрузке таблицы, Id сгенерируются автоматически от 1 до своего максимального значения.*

*//Если у вас 20 вещей в таблице, то сгенерируется от 1 до 20.*

*//Но при последующих добавлениях, id у добавленной вещи будет примерно таким - 345768.*

*//Я написал все так, чтобы последний мой id хранился в таблице и генерировался потом адекватно при последующих добавлениях.*

*//Также есть SEQUENCE, но он не поддерживается в Derby, а работать мы будем именно с ней.*

*//В общем, это нюансы. Можете узнать про них самостоятельно*

@TableGenerator(name = "cartid", table = "cartpktb", pkColumnName = "idCart",

pkColumnValue = "idCartValue",allocationSize = 1)

@GeneratedValue (strategy = GenerationType.TABLE, generator = "cartid")

private Integer id;

*//Указываем максимальный размер. Это строка из 25 символов.*

@Size(max = 25)

@Column(name = "NAMEITEM")

*//Если тип нашего поля String, то и создаваться будет столбец с типом VARCHAR( в Derby)*

*//Если Integer, то будет столбец, в который поместить можно только Integer*

*//boolean в Derby - это столбец с типом SMALLINT*

private String nameItem;

@Column(name = "PRICE")

private Integer price;

public CartEntity() {

}

public CartEntity( String nameItem, int price) {

this.nameItem = nameItem;

this.price = price;

}

public CartEntity(Integer id,String nameItem, int price) {

this.id=id;

this.nameItem = nameItem;

this.price = price;

}

public CartEntity(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getNameItem() {

return nameItem;

}

public void setNameItem(String nameItem) {

this.nameItem = nameItem;

}

public Integer getPrice() {

return price;

}

public void setPrice(Integer price) {

this.price = price;

}

}

#### Второй способ: пишем над геттерами. Пояснения смотрим в коде.

import java.io.Serializable;

import javax.persistence.\*;

*/\*\**

*\**

*\* @author miha*

*\*/*

@Entity

*//Видите, я не указывал аннотацию @Table*

*//Hibernate все поймет за меня.*

public class Product implements Serializable {

private Integer id;

private String nameProduct;

private Integer availableProduct;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

*//Я описал Id над геттером, значит, и с остальными полями работа будет идти через геттеры.*

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getNameProduct() {

return nameProduct;

}

public void setNameProduct(String nameProduct) {

this.nameProduct = nameProduct;

}

public Integer getAvailableProduct() {

return availableProduct;

}

public void setAvailableProduct(Integer availableProduct) {

this.availableProduct = availableProduct;

}

}

Итак, у Вас есть два примера классов сущностей. Создайте остальные, используя эти примеры. Такие поля как: модель, год публикации, имя, стоимость — все на Вашу фантазию. Включите обязательно поле Available(в переводе наличие). Оно Вам потом пригодится. Можете сделать его булевым и добавить столбец с именем количество. Это все нам пригодится.

### Теперь приведем пример нашего HibernateUtil

*/\**

*\* To change this template, choose Tools | Templates*

*\* and open the template in the editor.*

*\*/*

package edu.shumenko.hibernate;

import edu.shumenko.entity.BookEntity;

import edu.shumenko.entity.CableEntity;

import edu.shumenko.entity.CameraEntity;

import edu.shumenko.entity.CartEntity;

import edu.shumenko.entity.CustomerEntity;

import edu.shumenko.entity.NotebookEntity;

import edu.shumenko.entity.ProductEntity;

import org.hibernate.cfg.AnnotationConfiguration;

import org.hibernate.SessionFactory;

*/\*\**

*\* Hibernate Utility class with a convenient method to get Session Factory*

*\* object.*

*\**

*\* @author Mikhail Shumenko*

*\*/*

public class HibernateUtil {

*//Создаем нашу SessionFactory.*

private static final SessionFactory sessionFactory;

static {

try {

*//Создаем новый екземпляр AnnotationConfiguration*

AnnotationConfiguration ac = new AnnotationConfiguration();

*//Это нам нужно для того, чтобы мы добавили все наши классы сущности.*

*//каждый ваш Entity здесь нужно прописать, не пропишете - не будет работать.*

ac.addAnnotatedClass(ProductEntity.class).addAnnotatedClass(BookEntity.class).addAnnotatedClass(CableEntity.class)

.addAnnotatedClass(CameraEntity.class).addAnnotatedClass(NotebookEntity.class).

addAnnotatedClass(CartEntity.class).addAnnotatedClass(CustomerEntity.class);

*//Вот мы собственно и создали нашу Фабрику сессий.*

*//Она нужна т.к с БД мы работаем через сессии*

*//Подробности будут чуть позже, пока знайте, как ее сделать и как с ней работать.*

sessionFactory = ac.configure().buildSessionFactory();

} catch (Throwable ex) {

*// Log the exception.*

System.err.println("Initial SessionFactory creation failed." + ex);

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

return sessionFactory;

}

}

### DAO

Приступим к следующей части. Что такое DAO. Это шаблон проектирования.  
  
Его смысл:  
— Весь доступ к базе данных в системе производится через DAO для инкапсуляции.  
— Каждый экземпляр DAO отвечает за один первичный доменный объект или сущность. Если доменный объект имеет независимый цикл жизни, он должен иметь свой собственный DAO.  
— DAO отвечает за операции создания, чтения (по первичному ключу), обновления и удаления (то есть, CRUD (create, read, update, delete)) доменного объекта.  
— DAO может разрешать запросы, основанные на критерии, отличном от первичного ключа. Я ссылаюсь на такие методы как finder или finders. Метод finder обычно возвращает коллекцию доменных объектов, за которые отвечает DAO.  
— DAO не занимается обработкой транзакций, сессий или соединений. Это делается вне DAO для обеспечения гибкости.  
Подробнее всегда расскажет гугл.   
  
Мы пока напишем DAO для наших продуктов.  
Итак, подумаем что нам вообще нужно. Таблица Product будет иметь 4 поля Id,nameProduct,available+amount+actionForServlet. Она нам будет нужна, чтобы создать на нашем сайте категории. Над последним полем пока не заморачиваемся. Единственное что нам нужно — это получение списка продуктов.  
  
Пишем интерфейс

public interface ProductDAO {

ProductDAO INSTANCE\_PRODUCT= new ProductDAOImpl();

List<Product> getProducts();

*//и метод с которым мы будем работать*

}

Реализация нашего интерфейса. Пояснения смотрим в коде

*/\**

*\* To change this template, choose Tools | Templates*

*\* and open the template in the editor.*

*\*/*

package edu.shop.model.dao;

import java.util.List;

import org.hibernate.Criteria;

import org.hibernate.Session;

*/\*\**

*\**

*\* @author Mikhail Shumenko*

*\*/*

public class ProductDAOImpl implements ProductDAO {

@Override

public List<Product> getProducts() {

List<Product> result = null;

*//Создаем сессию, она нужна для использования транзакций*

*//Грубо говоря, транзакция - это как точка восстановления, если не прошла до конца, то все изменения откатываются.*

Session session = HibernateUtil.getSessionFactory().openSession();

try {

session.beginTransaction().begin();

*//Criteria используется для запроса с целью получения данных из БД*

*//Такой формулировки, думаю, Вам пока хватит*

*//Параметром мы передаем тот класс-сущность, который используем. Если бы данные получали из таблицы Cart то передавать*

*//надо было бы Cart.class*

Criteria criteria = session.createCriteria(Product.class);

result = (List<Product>) criteria.list();

session.getTransaction().commit

} catch (Exception e) {

*//Обработку исключений обязательно пишите. Но это я оставлю Вам на самостоятельную работу.*

e.printStackTrace();

}finally {

if (session != null) session.close();

}

return result;

}

}

Итак, теперь у нас есть возможность получать данные из БД. Вы можете, используя скриплеты, создать незамысловатый цикл for-each и вывести свою продукцию на вашу страницу index.jsp

<table border="2" width="2" cellspacing="2" cellpadding="2">

<thead>

<h3>Категории</h3>

</thead>

<tbody>

<tr>

<th>Категория</th>

//INSTANCE\_PRODUCT что это такое ?

//В ProductDAO описана такая переменная, отвечает за создание ProductDAOImpl

//Ну у нас все будет по-другому, можете особо не запоминать это.

//ProductDAO INSTANCE\_PRODUCT= new ProductDAOImpl();

<% for (Product product :ProductDAO.INSTANCE\_PRODUCT.getProducts()) {%>

<td><a href="<%=product.getActionForServlet()%>"><%=product.getName()%></a></td>

<%}%>

</tr>

<tr>

<th>Наличие</th>

<% for (Product product : ProductDAO.INSTANCE\_PRODUCT.getProducts()) {%>

<td><%=product.getAvailable()%></td>

<%}%>

</tr>

</tbody>

</table>

Но это на первое время, а вообще так делать плохо. Это нарушает наш паттерн MVC. Как сделать правильно я объясню в следующем уроке, если мне дадут инвайт. Во втором уроке мы займемся Spring, в третьем коснемся паттерна Factory, и углубимся в хибернейт. Для особо нетерпеливых, покажу как нужно удалять из БД, сохранять в БД и удалять полностью все из БД. Ну, а как сделать, чтобы все это взаимодействовало в целом с нашим приложением и подробное рассмотрение оставим на потом. 

#### Сохранить в БД

public void addItemToCart(CartEntity cart) {

Session session = HibernateUtil.getSessionFactory().openSession();

try {

session.beginTransaction().begin();

session.save(cart);

session.getTransaction().commit();

} catch (Exception ex) {

ex.printStackTrace();

} finally {

if (session != null) {

session.close();

}

}

}

#### Удалить из базы по id

public void deleteItemFromCart(Integer id) {

Session session = HibernateUtil.getSessionFactory().openSession();

try {

session.beginTransaction().begin();

CartEntity itemDelete = (CartEntity) session.get(CartEntity.class, id);

session.delete(itemDelete);

session.getTransaction().commit();

} catch (Exception ex) {

ex.printStackTrace();

} finally {

if (session != null) {

session.close();

}

}

}

#### Удаление из базы группы id .

public void deleteAllItemToCart(List<Integer> idAllItemsInCart) {

Session session = HibernateUtil.getSessionFactory().openSession();

try {

session.beginTransaction().begin();

for(Integer id:idAllItemsInCart){

CartEntity itemDelete = (CartEntity) session.get(CartEntity.class, id);

session.delete(itemDelete);

}

session.getTransaction().commit();

} catch (Exception ex) {

ex.printStackTrace();

} finally {

if (session != null) {

session.close();

}

}

}

Также Вам нужен будет файл настройки Hibernate. Создайте в Derby БД shop. Имя и пароль пользователя root и pass соответственно. Если не получится — не расстраивайтесь — я уделю еще этому время. 

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

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.dialect">org.hibernate.dialect.DerbyDialect</property>

<property name="hibernate.connection.driver\_class">org.apache.derby.jdbc.ClientDriver</property>

<property name="hibernate.connection.url">jdbc:derby:*//localhost:1527/shop</property>*

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">pass</property>

<property name="connection.characterEncoding">UTF-8</property>

<property name="hibernate.hbm2ddl.auto">update</property>

</session-factory>

</hibernate-configuration>

О том, как заполнять наши БД поговорим позже. Можете заполнить их вручную. Либо дождаться следующего урока.