Web Tools Review Note

1. Introdution

- Computer Networking: Networking is the practice of linking two or more computing devices together
- Network Protocol: Network protocols defines a language of rules and conventions for communication between network devices
- HTTP provides a standard for web browsers and servers to communicate
- IP addressed can be assigned statically or dynamically
- TCP/IP: Transmission Control Protocol/Internet Protocol
- The Domain Name System (DNS) distributes the responsibility for assigning domain names and mapping them to IP networks by allowing an authoritative server for each domain to keep track of its own changes
- TCP and UDP
    - Like the Transmission Control Protocol, UDP uses the Internet Protocol to actually get a data unit (called a datagram) from one computer to another
    - Unlike TCP, however, UDP does not provide the service of dividing a message into packets (datagrams) and reassembling it at the other end
    - Specifically, UDP doesn't provide sequencing of the packets that the data arrives in.

TCP VS UDP

- TCP: connected, guarantee that the data has arrived safely and correctly, but put a higher load on laptop.
- UDP: connectionless, not provide any guarantee that the data you send will ever reach its destination, but getting lower overhead
- web browser would pick a random TCP port from a certain range of port numbers, and attempt to connect to port 80 on the IP address of the web server

- FTP servers use TCP ports 20 and 21 to send and receive information

- The process of url accessing:

    - The client browser establishes a TCP/IP connection with the server
    - The browser sends a request to the server
    - The server sends a response to the client
    - The server closes the connection

- Http is connectionless, An `HTTP` transaction begins with a request from the client browser and ends with a response from the server

- A HTTP response concludes:
    - Protocol -- Status code -- Description
    - Response headers
    - Entity body
- Typical layers of a software system:
    - Presentation layer
    - Business logic layer
    - Data layer

2. Servlet

- Advantages of Servlet
    - Efficient
    - Convenient
    - Powerful
    - portable
    - inexpensive
    - Secure
    - Mainstream
- Servlets are Java programs that run on Web or application server, acting as a middle layer between requests coming from web browser or other HTTP clients
- The process:
    - Read the explicit data sent by the client
    - Read the implicit HTTP request data sent by the browser
    - Generate the results
    - Send the explicit data to the client
    - Send the implicit HTTP response data

Code Section:

```
public class CSVServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) t
hrows ServletException, IOException{
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) th
rows ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/jsp/CSVSearch.jsp").forward(request, respo
nse);
}
}
```

- When the servlet is first created, its init method is invoked, so init is where you put one-time setup code. After this, each user request results in a `thread` that calls the service method of the previously created instance
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service

How to deploy servlet in `web.xml` file:

```
<web-app>
    <servlet>
        <servlet-name>StoreServlet</servlet-name>
        <servlet-class>ShopStore.StoreServlet</servlet-class>
        <param-name>user</param-name>
        <param-value>username</param-value>
    </servlet>

    <servlet-mapping>
        <servlet-name>StoreServlet</servlet-name>
        <url-pattern>/store</url-pattern>
    </servlet-mapping>
</web-app>
```

- the system makes a single instance of your servlet and then creates a new thread for each user request

important functions:

```
getParameter()
getParameterNames()
getParameterValues()
```

Session Management

- the servlet sends a token or an identifier, while user requests a servlet, so next time the servlet will recognize this user
- Four ways to achieve session management:
  - URL Rewriting
  - Hidden Fields
  - Cookies
  - Session Objects

- Hidden Field

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">
```

- The main problem of session is the scalability of Session objects

- Some functions of Session:

```
HttpSession session = request.getSession(true);
session.getAttribute()
session.setAttribute()
session.getAttributeNames()
```

JSP

- JSP is a specification to create dynamic web pages based on the servlet specifications
- The JSP page is translated into a servlet and compiled only the first time it is accessed after having been modified
- JSP can convert into servlet after compiling
- The benefits of JSP over servlet
  - It is easier to write and maintain the HTML
  - You can use standard Web-site development tools
  - You can divide up your development team

- JSP VS Servlet

  - Similarities
    - Server-side execution
    - Provide identical results to the end user
    - JSP will convert to Servlet

  - Differences

    - Servlet: Mixes presentation with logic
    - JSP: Separates presentation from logic

- JSP Scripting Elements

```
<%= Java Expression %>
<% Java Code %>
<%! Field/Method Declaration %>
```

JavaBeans

- Beans are simply Java classes that are written in a standard format to expose data through properties
- Definition of JavaBean
    - Beans extend no particular class
    - are in no particular package
    - use no particular interface

- Advantages of JavaBean
    - No Java syntax
    - Simpler object sharing
    - Convenient correspondence between request parameters and object properties

- Characters of Bean
    - A bean class must have a zero-argument (default) constructor
    - A bean class should have no public instance variables
    - Persistent values should be accessed via getXxx and setXxx

- Using Beans

```
<jsp:useBean id=“beanName” class=“package.ClassName” scope=“scopeType” />
<jsp:getProperty name=“beanName” property=“propertyName”/>
<jsp:setProperty name="beanName“ property="propertyName“ value="propertyValue" />
```

- A instance for JavaBean:

```
<jsp:useBean id="myCar" class="com.yusuf.cars.CarBean" />
<html>
    <head>
    <title>Using a JavaBean</title>
</head>
<body>
<h2>Using a JavaBean</h2>
I have a <jsp:getProperty name="myCar" property="make" /><br> <jsp:setProperty na
me="myCar" property="make" value="Ferrari" /> Now I have a <jsp:getProperty name=
"myCar" property="make" />
    </body>
</html>
```

- We can get javabean from 4 different sessions:
    - page
    - request
    - session
    - application
- Description of Model Architecture
    - Model 1 Architecture: No Servlet
    - JSP Model 2 Architecture: model-view-controller (MVC)
- Required Steps in MVC
    - Define beans to represent the data
    - Use a servlet to handle requests
    - Populate the beans
    - Store the bean in the request, session, or servlet context
    - Forward the request to a JSP page
    - Extract the data from the beans
- Difference between forword and redirect
    - With forward, there is no extra response/request pair as with sendRedirect, thus url don't change
- Summary of the behavior of forward
    - Control is transferred entirely on the server
    - No network traffic is involved
    - The user does not see the address of the destination JSP page and pages can be placed in WEB-

INF to prevent the user from accessing them without going through the servlet that sets up the data

- Summary of the behavior of sendRedirect
  - Control is transferred by sending the client a 302 status code and a Location response header
  - Transfer requires an additional network round trip
  - The user sees the address of the destination page

Spring MVC

- We should configure `dispatcher-servlet.xml` into `web.xml`, like:

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-cla
ss>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- Three Different Handler Mapping
  - BeanNameUrlHandlerMapping

```
<bean name=“handlerMapping” class=“org.springframework.web.servlet.handler.Be
anNameUrlHandlerMapping”/>
<bean name=“/editaccount.htm” class=“AccountController”></bean>
```

  - SimpleUrlHandlerMapping

```
<bean id="simpleurlhandlemapping" class="org.springframework.web.servlet.hand
le.SimpleUrlHandlerMapping">
    <property>
        <props>
            <prop key="url inside">(id of controller)<prop>
            <prop key="add/movie">addMovieController</prop>
        </props>
    </property>
</bean>
<bean id = "addMovieController" class = "MovieSection.AddController"/>
```

- ControllerClassNameHandlerMapping(according to the class name to distinguish path)----ClassNameMovie, ClassNameAdd, ClassNameSearch are url

```
<bean class = "org.springframework.web.servlet.mvc.support.ControllerClassNam
eHandlerMapping"/>
<bean class = "MovieSection.ClassNameSection.ClassNameMovieController" />
<bean class = "MovieSection.ClassNameSection.ClassNameAddController"/>
<bean class="MovieSection.ClassNameSection.ClassNameSearchController" />
```

- In mapping beans, we can define some interceptors to deal with specific issues, like:

```
public class BigBrotherHandlerInterceptor extends HandlerInterceptorAdapter
{
    public void postHandle(HttpServletRequest request, HttpServletResponse respon
se, Object handler, ModelAndView modelAndView) throws Exception {
    }
}
```

in dispatcher-web.xml, we can define like this:

```
<property name="interceptors">
    <list>
    <ref local="bigBrotherHandlerInterceptor"/>
    </list>
</property>
```

Dependency Injection

Two important terms:

- Inversion of Control (IoC): objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside sources
- Dependency Injection (DI): this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties
- IoC and DI aim to solving the problem of tight coupling
- About the life cycle of the service:

> without the IoC, the class is responsible for handling the life-cycle of the service. Using the concept of dependency injection, however, the life-cycle of a service is handled by a dependency provider. Cosumer only need the `reference`

- DI includes:
  - dependent
  - dependencies
  - injector
- Three different way to `dependency injection` :
  - interface injection
  - setter injection
  - constructor injection
- Setter Injection(most often)

```
<beans>
    <bean id="Movie" class="">
        <constructor-arg value=""></constructor-arg>
        <property name="DAOsetion">
            <ref local="movieDAO" />
        </property>
    <bean/>
    <bean id="movieDAO" class="">
        <property>
            <value></value>
        <property>
    </bean>
</beans>
```

- How to inject in Java Class:

```
// Like this:
movieDAO moviedao = (movieDAO) getApplicationContext().getBean("movieDAO");
```

FormController

- simpleFormController
  - Set commandClass and commandName in constructor
  - Set formView, successView, validator
  - override onSubmit function

Just know this simpleFormController

```
public class addMovieController extends SimpleFormController {
    public addMovieController() {
        this.setCommandClass(MovieEntity.class);
        this.setCommandName("MovieEntity");
    }

    @Override
    protected void onBindAndValidate(HttpServletRequest request, Object command, Bind
Exception errors) throws Exception {
        super.onBindAndValidate(request, command, errors);
    }

    @Override
    protected ModelAndView onSubmit(Object command, BindException errors) throws Exce
ption {
        MovieEntity movieObj = (MovieEntity) command;
        return new ModelAndView(this.getSuccessView());
    }
}
```

In dispatcher-servlet.xml, we should define like this:

```
<bean id = "addMovieController" class = "movieController.addMovieController">
    <property name="validator" ref="movieValidator"/>
    <property name="formView" value="AddMovieJSP"/>
    <property name="successView" value="MovieJSP"></property>
</bean>
```

In validator, the only thing we need to do is to override `validate()` function

```
public class movieValidator implements Validator {
    @Override
    public boolean supports(Class clazz) {
        return clazz.equals(MovieEntity.class);
    }
    @Override
    public void validate(Object obj, Errors err) {
        ValidationUtils.rejectIfEmptyOrWhitespace(err, "title","title.required", "tit
le invalid");
    }
}
```

Hibernate

- Persistence: The storage of data, in object-oriented systems, is called persistence
- Persistence Mechanisms
    - direct use of JDBC
    - ORM(object-to-relational-mapping)
    - the implementation of a unifying data access object (DAO) abstraction layer(?? I don't know)
- `Hibernate` is one of the `ORM` framework, mapping the relational model to OO model is done outside of the language, typically in xml files
- Mapping

> This step is to map POJO class to table, and this is one of the most important steps while using Hibernate, if we define a class like following one:

```
public class message{
    private int id;
    private String title;
    ..... getter/setter.....
}
```

Then we can define `message.hbm.xml` as following, in this file we need to map the POJO to table:

```
<hibernate-mapping>
    <class name="the class name" table="the table name">
        <id name="id" type="java.lang.Integer">
            <column name="id"/>
            <generator class="native"></generator>
        </id>
        <property name="title" type="java.lang.String">
            <column name="title"></column>
        </property>
    </class>
</hibernate-mapping>
```

In mapping file, the every element in POJO, you can three elements to define: column_name, type, name(of course the name in POJO class)

After defining mapping file, we should define `Hibernate Configuration file`, named `hibernate.cfg.xml`. In this file, we define the relevant parameters, which are used to connect JDBC, and

also add mapping file into this file:

```
<hibernate-configuration>
    <session-factory>
        <property name="connection.url"></property>
        <property name="connection.driver_class">....</property>
        username, password, and other elements
        <mapping resource="message.hbm.xml"></mapping>
    </session-factory>
</hibernate-configuration>
```

Then it is one of the most important steps to instantiate a session factory, and there are two different ways to achieve that:

a.

```
Configuration configuration = new Configuration();
configuraion.configure("hibernate.cfg.xml");
try{
    SessionFactory sessionfactory = configuration.buildSessionFactory();
}catch(Exception e){
    System.out.println(e.getMessage());
}
```

b. (I won't remeber that, it is too long, you decide it)

```
try{
StandardServiceRegistry standardRegistry = new StandServiceRegistrybuilder().configur
e("hibernate.cfg.xml").build;
Metadata metaData = new MetadataSources(standardRegistry).getMetadataBuilder().build(
);
sessionFactory = metaData.getSessionFactoryBuilder().build();
}catch(Exception e){
    .....
}
```

- We achieve two different operations on hibernate, `add` and `search`, after creating `sessionFactory`:

  ```
  Session session = sessionfactory.openSession();
  session.save(object of POJO);
  session.getTransaction().commit();
  ```

- Achieving `search` function:

```
Session session = sessionfactory.openSession();
q = Session.createQuery("from movie where actor=:actor");
q.setString("actor", "value");
(movieDAO) result = q.list();
session.getTransaction().commit();
```

Mapping Collection

- Entity vs Value
    - Entity
        - Entities have IDs
        - Entities live in there own spaces
        - Entities could be shared
        - Entities could exist on their own.
    - Value
        - Values don't have IDs
        - Values generally live in a space belonging to the owner, typically in a entity
        - values cannot be shared
        - values cannot exist on their own
- The lifespan of a value-type instance is bounded by the lifespan of the owning entity instance.
- The difference between Mapping collection and mapping general hibernate is the definition of mapping file:

Java Collection	Initialize	tag element
java.util.Set	java.util.HashSet	<set>
java.util.SortedSet	java.util.TreeSet	<set>
java.util.List	java.util.ArrayList	<list>
java.util.Collection	bag collection	<bag> /<idbag>

- How to map a set(heat!!):

```
<set name="the element name in class" table="table name">
    <key column="tag the user or main object">
    <element column="The column name in table" not-null="true" type=""/>
<set>
```

- How to map a list:

> list has postion, thus in table we must define postion, by using `list-index`

```
    <list name="" table="">
        <key column="" />
        <list-index column="position"/>
        <element column="" type="" not-null="" />
    </list>
```

- How to map a Map:

```
<map name="" table="">
    <key column="" />
    <map-key column="" type=""/>
    <element column="" type="" not-null="true"/>
</map>
```

- How to map a sorted or ordered collection:

> Using the `sort` session

SortedMap

```
<map name="" sort="natural">
    <key colum=""/>
    <map-key column="" type="" />
    <element column="" type="" not-null="" />
</map>
```

If we want to achieve special sort method, we should use comparator like:

> <map name="" sort="MyComparator">

```
public class MyComparator implements Comparator<Address>{
        public int compare(Address o1, Address o2){
            if(o1,zip > o2.zip)
                return 1;
            else if(o1.zip < o2.zip)
                return -1;
            else
                return 0;
        }
    }
```

- How to map a collection(<bag>), collecion is special, there is a tag name `collection-id` needing to assgin:

```
<ibag name="" table>
    <collection-id column="" type="">
        <generator class="sequence">
    </collection-id>
    <key column=""/>
    <element column="" type="" not-null="true"/>
</ibag>
```

- If we want to order by certain variables, we can use:

```
<map name="images“ table="ITEM_IMAGE“ order-by="IMAGENAME asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

    - You can also use SQL function in `order-by` attribute:

> order-by="lower(FILENAME) asc"

Entities

- The purpose of Hibernate is used to store Java objects into databases
- So, the entities chapter mainly focus on how to store objects into databases, and how to define the relationship among different entities
- In Hibernate, every table must require a `primary key`. Without primary key, it is impossible to uniquely identify a row in table

- `Lazy loading` : Certain relationships can be marked as being “lazy,” and they will not be loaded from disk until they are actually required
    - By default, the classes(including collection lieke set, map) are lazily loaded, which means like follwing code:

```
public class User {
    int userId;
    String username;
    EmailAddress emailAddress;
    Set roles;
}
```

    - Only userId and username is initialized into memory, others will be loaded when requires

Associations

- This term is used to define the relationships between different entities
    - One-to-one
    - One-to-many
    - Many-to-one
    - Many-to-many

- One-to-one : `<one-to-one>`
- Reference: [How to define one-to-one relationship in Hibernate]

    - Each instance of the first class is related to a single instance of the second, and vice versa.
    - **Notice: We must define the `foreign key` in slave class in order to connect the entities with each other**
    - For Employee

```
empoyee section
<hibernate-mapping>
    <class name="com.yiibai.Employee" table="emp_2120">
        <id name="employeeId">
            <generator class="increment"></generator>
        </id>
        <property name="name"></property>
        <property name="email"></property>

        <one-to-one name="address" class="" constrained="true" cascade="all">
</one-to-one>
    </class>
</hibernate-mapping>
```

- For Address

```
<hibernate-mapping>
    <class name="com.yiibai.Address" table="address_2120">
        <id name="addressId">
            <generator class="foreign">
                <param name="property">employee</param>
            </generator>
        </id>
        <property name="addressLine1"></property>
        <property name="city"></property>
        <property name="state"></property>
        <property name="country"></property>

        <one-to-one name="employee" class="" constrained="true" cascade="all"
></one-to-one>
    </class>
</hibernate-mapping>
```

- Many-to-one: `<many-to-one>`
  - The many-to-one association describes the relationship in which multiple instances of one class can reference a single instance of another class
  - Reference: How to define many-to-one in Hibernate
  - We suppose that many employees may have same address, then how to define employee

```
<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">

      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="Name" column="name" type="string"/>
      <property name="salary" column="salary" type="int"/>
      <many-to-one name="address" column="address"
       class="Address" not-null="true" unique="true"/>
   </class>
</hibernate-mapping>
```

- How to define address

```
<hibernate-mapping>
    <class name="Address" table="ADDRESS">
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="street" column="street" type="string"/>
      <property name="city" column="city_name" type="string"/>
      <property name="state" column="state_name" type="string"/>
      <property name="zipcode" column="zipcode" type="string"/>
   </class>
</hibernate-mapping>
```

- Another way to define one-to-many is to use `<join>`
- Reference: How to use to define the relationship
- In class we may define the collection of classes, like:

```
set<Employee> employee;
List<Address> addresses;
```

- We have to map such relationship as reverse association:

```
<set name="phoneNumbers" inverse="true">
    <key column="employee"/>
    <one-to-many class="Employee"/>
</set>
```

    - Reference: How to define the collection of classes in Hibernate

- Many-to-many: `many-to-many`
    - Reference: How to define many-to-many relationship
    - We can define category-item as many-to-many relationship, category is like:

```
<hibernate-mapping>
    <class name="category" table="categoryTable">
        <id name="categoryid" type="java.lang.Integer">
            <column name="">
         <generator class="native"/>
      </id>
      <set name="items" table="category_item">
        <key column="categoryid"/>
        <many-to-many class="" column="itemid"/>
      </set>
    </class>
</hibernate-mapping>
```

    - Also, we can define `item` as:

```
<hibernate-mapping>
    <class name="item" table="itemTable">
        <id name="itemid" type="java.lang.Integer"/>
            <column name="" />
            <generator class="native"/>
        </id>
        <set name="category" table="category_item" inverse="true">
            <key column="itemid"/>
            <many-to-many class="category" column="categoryid"/>
        </set>
    </class>
</hibernate-mapping>
```

    - Notice: We must set `inverse`, which means that the relationship is sychronized with `item` collection

Hibernate Searches and Queries

- In Hibernate, we sue Hibernate Query Language(HQL) to phrase these requests, which can achieve extract, insert, update and delete

- HQL queries are translated by Hibernate into conventional SQL queries; Hibernate also provides an API that allows you to directly issue SQL queries
- HQL was inspired by SQL and is the inspiration for the Java Persistence Query Language (JPQL)
- Upadate Operation

```
Query query = session.createQuery("update Person set creditescore=:creditscore wh
ere name=:name")
query.setInteger("creditscore", 166);
query.setString("name","xxxx");
int result = query.excuteUpdate();
```

- Delete Operation

```
Query query = session.createQuery("delete from person where name=:name");
query.setString("name", "xxx");
int result = query.excuteUpdate();
```

- Insert Operation

```
Query query = session.createQuery("insert into purged_user(id, name, status) " +
"select id, name, status from user where name=:name");
// Query query = session.createQuery("insert into purger_user(id, name, status) v
alues(xx, xx, xx)");
query.setString("name", "xxx");
int result = query.excuteUpdate();
```

- Select Operation
  - Most simple way

```
query = session.creteQuery("from supplier");
List result = query.list();
```

  - The point is how to build the HQL

```
String hql = "from Product where price > :price";
Query query = session.createQuery(hql);
query.setDouble("price",25.0);
List results = query.list();
```

  - How to extract unique result

```
String hql = "from Product where price>25.0";
Query query = session.createQuery(hql);
query.setMaxResults(1);
Product product = (Product) query.uniqueResult();
```

- `order by` Clause

```
from Product p where p.price > 25.0 order by p.price desc
// or
from Product p order by p.supplier.name asc, p.price asc
```

- Associations
  - To join different tables to extract data

```
select s.name, p.name, p.price from Product p inner join p.supplier as s
```

- Aggregate Methods

```
select count(*) from Product product
//
select count(distinc p.supplier.name) from Product p
```

Annotations

- An annotation, in the Java computer programming language, is a special form of syntactic metadata that can be added to Java source code
- Classes, methods, variables, parameters and packages may be annotated
- Annotations provide data about a program that is not part of the program itself
- They have no direct effect on the operation of the code they annotate
- The functions of Annotations
  - Information for the compiler
  - Compiler-time and deployment-time processing--(Software tools can process annotation information to generate code, XML files, and so forth)
  - Runtime processing
- Three basic annotations
  - @Override(override parent's function)
  - @Deprecated
  - @SuppressWarnings: to suppress some warning

        - @SuppressWarnings("unchecked")
        - @SuppressWarnings(value={"unchecked", "rawtypes"})

- Mapping with Annotations

```
@Entity
public class Sample{
    @Id
    private Integer id;
    private String name;
    // getter() and setter()
}
```

Annotations in Hibernate

- Hibernate Model
    - Hibernate Core: offers native API's & object/relational mapping with XML metadata
    - Hibernate Annotations
    - Hibernate EntityManager

- What is JPA? JPA is POJO API for ORM that supports not only Java metadata annotations and XML metadata
- Annotations are embedded in class files generated by compiler
- Hibernate annotations can be divided into two ways: basic annotations, Hibernate extension annotations
- Basic Annotations
    - @Entity
    - @Id
    - @EmbeddedId
    - @GeneratedValue
    - @Table
    - @Column
    - @OneToOne
    - @ManyToOne
    - @OneToMany

- Reference: How to use Hibernate Annotations
- The keynote of Hibernate Annotations is

    - you should set annotations before `get() function` to map certain column
    - We should know how to make annotations to mark id generating

```
@GenericGenerator(name = "generator", strategy = "increment")
@Id
@GeneratedValue(generator = "generator")
@Column(name = "ID", unique = true)
```

- There is the example that how to use Hibernate Annotations:

```
@Entity
@Table(name = "CATEGORY")
public class Category{
private Long id;
private String name;
private Set<Product> products = new HashSet<Product>(0);

    public Category() {
    }

    // Property accessors
    @GenericGenerator(name = "generator", strategy = "increment")
    @Id
    @GeneratedValue(generator = "generator")
    @Column(name = "ID", unique = true, nullable = false, precision = 8, scale =
0)
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "NAME", length = 400)
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
    @OneToMany(mappedBy = "category")
    public Set<Product> getProducts() {
        return this.products;
    }

    public void setProducts(Set<Product> products) {
        this.products = products;
    }

}
```

- In one-to-many Annotations, `mappedBy` is used to point out which one is in charge of maintaining the

relationship

Hibernate Criteria

- Criteria is another way to retrieve data from databases
- Criteria API allowes people to build up a criteria query object
- The simplest query of Criteria is shown as follow:

```
Criteria crit = session.createCriteria(Product.class);
List<Product> results = crit.list();
```

- We can use `add()` to add any restriction, and you can add more than one restriction for the Criterion objects
- We can nest `eq()` function on `Restrictions` to make restriction of the objects

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("description",“Tablet"));
List<Product> results = crit.list();
```

- If we want to find **not Tablet**, we can use `ne()` function to achieve that

```
.....
crit.add(Restrictions.ne("description",“Tablet"));
.....
```

- If we want to match the string or not match the string
  - we can use function `like()` and `ilike()` to achieve:

    ```
    Criteria crit = session.createCriteria(Product.class);
    crit.add(Restrictions.like("name",“Tab%"));
    List<Product> results = crit.list();
    ```

  - If we want to match the string in a more efficient way, we can use some parameters to achieve that:

    ```
    crit.add(Restrictions.ilike("description","ser", MatchMode.END);
    ```

    - ANYWHERE: Anyplace in the string
    - END: The end of the string
    - EXACT: An exact match

    - START: The beginning of the string

- `isNull()` and `isNotNull()`

```
crit.add(Restrictions.isNull("name"));
```

- Some compare function

    - gt(): great than
    - ge(): great than or equal to
    - lt(): less than
    - le(): less than or equal to

    ```
    crit.add(Restrictions.gt("price", 25.0));
    ```

- If we want to add more than one restriction to achieve `AND` restrictions, we just need to use `add()` to add every restriction

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.lt("price",10.0));
crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

- If we want to express `OR` operation, we can use `or()` function to combine to restrictions. **Notice**: we can use `LogicalExpression` to mark the restrictions:

```
Criteria crit = session.createCriteria(Product.class);
Criteron lessthan = crit.add(Restricions.lt("price",15));
Criteron tablet = crit.add(Restrictions.like("description","tablet", MatchMode.EX
ACT));
LogicalExpression orExp = Restrictions.or(lessthan, tablet);
crit.add(orExp);
List<Product> result = crit.list();
```

- If we want to express more than one restrictions to make `OR` / `AND` operations, we can use:

    - disjunction() in Restrictions: `OR` --- Disjunction
    - conjunction() in Restrictions: `AND` --- Conjunction

```
Criteria crit = sesssion.createCritera(Product.class);
Criteron priceLessThan = Restrictions.lt("price", 10.0);
Criteron mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHER
E);
Criterion browser = Restrictions.ilike("description", "browser", MatchMode.AN
YWHERE);
Disjunction disjunction = Restrictions.disjunction();
disjunction.add(priceLessThan);
disjunction.add(mouse);
disjunction.add(browser);
crit.add(disjunction);
List<Product> result = crit.list();
```

- If you think Critera is not efficient enough to make restrictions, you can make restrictions directly by using `sqRestriction()` function:

```
crit.add(Restrictions.sqRestriction("{tableName}.description like 'table%'"));
```

- We can implement pagination to relif the burdon of memory, mainly use two functions as follows:
  - setFirstResult(): set first row
  - setMaxResult(): set how many result you want to acquire

```
Criteria crit = session.createCriteria(Product.class);
crit.setFirstResult(1);
crit.setMaxResults(20);
List<Product> results = crit.list();
```

- How to get unique result, we can set: `crit.setMaxResults(1);` , and if we just want to get certain one object instead of the List, we can operate as follows:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
crit.setMaxResults(1);
Product product = (Product) crit.uniqueResult();
```

- if we want to make `order by` operation, we can operate like:
  - crit.addOrder(Order.desc("price"));
  - crit.addOrder(Order.asc("xxxx"));
- one-to-many associations

    - one supplier has more than one products
    - We obtain supplier according to the restriction of product

```
Criteria crit = session.createCriteria(Supplier.class);
Criteria procit = crit.creatCriteria(Product.class);
procit.add(Restrictions.bt("price", 25));
List<Supplier> result = crit.list();
```

- For the former example, in verse, it is `many-to-one` association:

```
    Criteria crit = session.createCriteria(Supplier.class);
    Criteria procit = crit.creatCriteria(Product.class);
    procit.add(Restrictions.bt("name", "Leo"));
    List<Product> result = crit.list();
```

- We can use the functions in `Projections` to acquire certain result:

```
Criteria crit = session.createCrieria(Product.class);
crit.setProjection(Projections.rowCount());
List<Long> result = crit.list();
// result has one Long representing row count
```

- If we want to add more than one projection, we can use `projectionList()`, like:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.max("price"));
projList.add(Projections.min("price"));
projList.add(Projections.countDistinct("description"));
crit.setProjection(projList);
List<Object[]> results = crit.list();
```

- If we just want to get some properties among the table instead of all attributes, we can use `Projections.property("xxx")` to achieve that:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.property("xxxx"));
projList.add(Projections.property("xxxx"));
crit.setProjection(projList);
List<object[]> result = crit.list();
```

- One of the most important operations in SQL is `GROUP`, like:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.groupPrperty("name"));
projList.add(Projections.groupPropery("price"));
crit.setProjection(projList);
crit.addOrder(Order.desc("price"));
List<object[]> result = crit.list();
```

- Example: 不看了，考了就得0分吧