Лабораторная №8

Spring Framework.

**Spring Annotation, Hibernate**.

**Цель**: Разработать учебную подсистему чтения и сохранения в базе данных информации о клиентах условного сервиса используя JAXB и Hibernate.

**Среда**. OS - произвольно. Рекомендуемый язык программирования Java. Рекомендуемые базы данных – MySQL/MariaDB.

**Краткие теоретические сведения**

*Полезные ссылки:*

*http://www.oracle.com/technetwork/articles/javase/index-140168.html http://hibernate.org/*

**Что такое JDBC?**

JDBC (Java Database Connectivity) – это технология, которая обеспечивает доступ Java API к реляционным базам данных. Благодаря этому, наши Java-приложения могут выполнять SQL-запросы и взаимодействовать с базами данных (далее – БД), котрые поддерживают SQL.

JDBC является крайне гибкой и позволяет нам писать приложения, которые не зависят от конкретной платформы и могут взаимодействовать с различными СУБД без каких либо изменение в программном коде.

Какие плюсы даёт нам JDBC:

* + Простая и понятная обработка SQL-запросов
  + Крайне убдобна для небольших приложений

Простой и понятный синтаксис

Какие минусы JDBC:

* Сложно использовать и поддерживать в больших проектах
* Большое количество кода
* Сложно реализовывать концепцию MVC.

**Что такое ORM (Object Relational Mapping)?**

Когда мы создаём приложение на Java и хотим сделать так, чтобы наше приложение получило доступ к информации, которая хранится в БД, мы должны понимать крайне важную деталь.

**Сущействует огромная разница между объектной моделью и реляционной.**

СУБД даёт нам информацию в табличном формате, в то время, как Java даёт нам информацию в виде некоего графа объектов.

Отсюда возникает проблема: как сделать так, чтобы Java приложение получало доступ к БД и могло корректно интерпретировать эту информацию.  
Другими словами, нам нужно создать связь между Объектом и реляционной сущностью, иначе говоря Объектно-Реляционное-Связывание или же – **ORM (Object-Relational Mapping).**

ORM – это техника программирования, которая служит для того, чтобы обеспечивать преобразование данных при их обмене между реляционной базой данных и (в нашем случае) Java.

Так какие же преимущества нам даёт ORM в сравнение с JDBC?

* Позволяет нашим бизнес методам обращаться не к БД, а к Java-классам
* Ускоряет разработку приложения
* Основан на JDBC
* Отделяет SQL-запросы от ОО модели
* Позволяет не думать о реализации БД
* Сущности основаны на бизнес-задачах, а не на стуктуре БД
* Управление транзакциями

ORM состоит из:

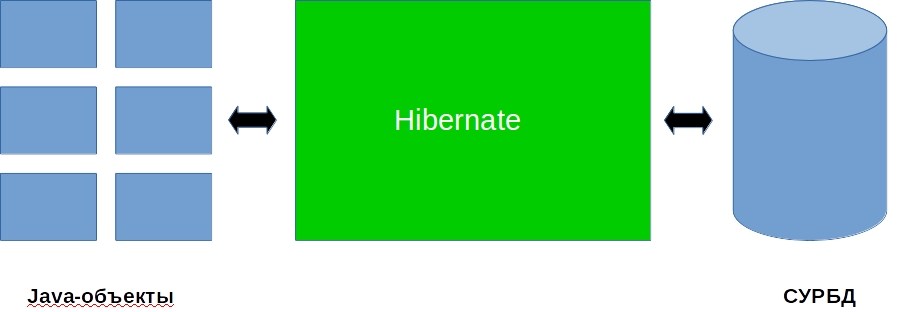
* API, который реализует базовые операции (СОЗДАНИЕ, ЧТЕНИЕ,  ИЗМЕНЕНИЕ, УДАЛЕНИЕ) объектов-моделей.
* Средства настройки метаданных связывания
* Технику взаимодействия с транзакциями, которая позволяет реализовать такие функции, как dirty checking, lazy association fetching и т.д.

А самыми распрастранёнными ORM фреймворками являются:

* Hibernate (который мы и будем изучать)
* EJB
* JDO
* Spring DAO
* TopLink

Hibernate создаёт связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот. Это избавляет разработчиков от огромного количества лишней, рутинной работы, в которой крайне легко допустить ошибку и крайне трудно потом её найти..

Схематично это можно изобразить следующим образом:



Средствами утилит Hibernate можно создать классы, соответствующие таблицам БД. В результате будут сгенерированы классы (JavaBean) для каждой из таблиц.

Класс model:

package net.hibernate.example.model;  
  
public class Developer {  
 private int id;  
 private String firstName;  
 private String lastName;  
 private String specialty;  
 private int experience;  
public Developer() {  
 }public Developer(String firstName, String lastName, String specialty, int experience) {  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.specialty = specialty;  
 this.experience = experience;  
 }  
public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getSpecialty() {  
 return specialty;  
 }  
  
 public void setSpecialty(String specialty) {  
 this.specialty = specialty;  
 }  
  
 public int getExperience() {  
 return experience;  
 }  
  
 public void setExperience(int experience) {  
 this.experience = experience;  
 }  
  
 */\*\*  
 \* toString method (optional)  
 \*/* @Override  
 public String toString() {  
 return "Developer:\n" +  
 "id: " + id +  
 "\nFirst Name: " + firstName + "\n" +  
 "Last Name: " + lastName + "\n" +  
 "Specialty: " + specialty + "\n" +  
 "Experience: " + experience + "\n";  
 }  
}

Developer.hbm.xml для маппинга с БД.

<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class name="net.hibernate.example.model.Customer" table="HIBERNATE\_ CUSTOMER">  
 <meta attribute="class-description">  
 This class contains customer’s details.  
 </meta>  
 <id name="id" type="int" column="ID">  
 <generator class="native"/>  
 </id>  
 <property name="firstName" column="FIRST\_NAME" type="string"/>  
 <property name="lastName" column="LAST\_NAME" type="string"/>  
 <property name="specialty" column="SPECIALTY" type="string"/>  
 <property name="experience" column="EXPERIENCE" type="int"/>  
 </class>  
</hibernate-mapping>

Для упрощения взаимодействия можно создать HibernateUtil, который возвращает SessionFactory.

public class HibernateUtil {  
 private static final SessionFactory *sessionFactory*;  
  
 static {  
 try {  
 *sessionFactory* = new AnnotationConfiguration().configure().buildSessionFactory();  
 } catch (Throwable ex) {  
 System.*err*.println("Initial SessionFactory creation failed." + ex);  
 throw new ExceptionInInitializerError(ex);  
 }  
 }  
  
 public static SessionFactory getSessionFactory() {  
 return *sessionFactory*;  
 }  
}

**2.1. Чтение данных из БД с использованием Hibernate**

Session session=null;Transaction tx=null;  
 StringBuilder sb=new StringBuilder();try{  
 session=HibernateUtil.getSessionFactory().openSession();tx=session.beginTransaction();  
 List list=session.createQuery("FROM History").list();  
// Запрос к БД  
 sb.append("Всего куплено товаров: ").append(list.size()).append("<br><br>");  
 for(Iterator iterator=list.iterator();iterator.hasNext();)  
 {  
 History cur=(History)iterator.next();sb.append("<b>Название товара:  
</b>").append(cur.getProduct());  
 sb.append("<b>Магазин: </b>").append(cur.getMarket());  
 sb.append("<br>");  
 }  
 tx.commit();  
 }catch(HibernateException e){if(tx!=null)  
 tx.rollback();  
 e.printStackTrace();  
 }finally{  
 session.close();  
 }

**2.2. Запись данных в БД с использованием Hibernate**

Session session=NewHibernateUtil.getSessionFactory().openSession();Transaction tx=null;  
 StringBuilder sb=new StringBuilder();Integer resId=null;  
 try{  
 tx=session.beginTransaction();  
 History h=new History(marketM,productQ);resId=(Integer)session.save(h);  
//Сохранение нового объекта tx.commit();  
 sb.append("<br>Название товара: ").append(h.getProduct());sb.append(";");  
 sb.append("Магазин: ").append(h.getMarket()).append(";<br>");  
 }catch(HibernateException e){  
 if(tx!=null)tx.rollback();  
 e.printStackTrace();}finally{  
 session.close();  
 }

# Что такое Spring?

Для начала пара слов, что же такое Spring. В настоящее время, под термином "Spring" часто подразумевают целое семейство проектов. В большинстве своем, они развиваются и курируются компанией Pivotal и силами сообщества. Ключевые (но не все) проекты семейства Spring это:

• Spring Framework (или Spring Core)

Ядро платформы, предоставляет базовые средства для создания приложений — управление компонентами (бинами, beans), внедрение зависимостей, MVC фреймворк, транзакции, базовый доступ к БД. В основном это низкоуровневые компоненты и абстракции. По сути, неявно используется всеми другими компонентами.

• Spring MVC (часть Spring Framework)

Стоит упомянуть отдельно, т.к. мы будем вести речь в основном о веб-приложениях. Оперирует понятиями контроллеров, маппингов запросов, различными HTTP абстракциями и т.п. Со Spring MVC интегрированы нормальные шаблонные движки, типа Thymeleaf, Freemaker, Mustache, плюс есть сторонние интеграции с кучей других. Так что никакого ужаса типа JSP или JSF писать не нужно.

• Spring Data

Доступ к данным: реляционные и нереляционные БД, KV хранилища и т.п.

• Spring Cloud

Много полезного для микросервисной архитектуры — service discovery, трасировка и диагностика, балансировщики запросов, circuit breaker-ы, роутеры и т.п.

• Spring Security

Авторизация и аутентификация, доступ к данным, методам и т.п. OAuth, LDAP, и куча разных провайдеров.

• Spring Integration

Обработка данных из разных источников. Если надо раз в час брать файл с ФТП, разбивать его на строки, которые потом фильтровать, а дальше отправлять в какую-то очередь — это к Spring Integration.

В данной лабораторной работе мы познакомимся ближе со Spring MVC и Spring Security.

**Плюсы Spring**:

• Spring можно использовать для построение любого приложения на языке Java, что выгодно отличает его от многих других платформ (таких как Apache Struts)

• для использования ядра Spring нужно внести минимальные изменения в код приложения (принцип философии Spring – минимальное воздействие)

• Spring является модульной средой и позволяет использовать отдельные свои части без необходимости вводить остальные

• Возможность работы с POJO (без контейнеров EJB)

• Существует большое количество расширений Spring для построения приложений на Java Enterprise платформе

• Сообщество Spring – одно из лучших сообществ из всех проектов с открытым исходным кодом, списки рассылки и форумы всегда активны.

• У Spring отличная подробная документация • Spring активно развивается

**Spring** обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе [Java](https://ru.wikipedia.org/wiki/Java). Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. **Spring** не всецело связан с платформой [Java Enterprise](https://ru.wikipedia.org/wiki/J2EE), несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности.

**Spring**, вероятно, наиболее известен как источник расширений (features), нужных для эффективной разработки сложных бизнес-приложений вне тяжеловесных программных моделей, которые исторически были доминирующими в промышленности. Ещё одно его достоинство в том, что он ввел ранее неиспользуемые функциональные возможности в сегодняшние господствующие методы разработки, даже вне платформы Java.

Этот фреймворк предлагает последовательную модель и делает её применимой к большинству типов приложений, которые уже созданы на основе платформы Java. Считается, что **Spring** реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java.

**Возможности Spring:**

• Использование внедрения зависимостей (DI)

• Поддержка аспектно-ориентированного программирования (в том числе интеграция с AspectJ)

• Язык выражений Spring (SpEL) – позволяет приложению манипулировать объектами Java во время выполнения.

• Встроенная поддержка Bean Validation API – позволяет один раз описать логику проверки достоверности данных и использовать ее как в пользовательском интерфейсе, так и на уровне работы с БД.

• Spring обеспечивает великолепную интеграцию с большинством инструментов доступа к данным (JDBC, Hibernate, MyBatis, JDO, JPA и т.п.)

• Поддержка Object to XML Mapping – преобразование компонентов JavaBean в XML и наоборот (как правило, используется для обмена данными с другими системами)

• Интеграция с JEE – внедрение бинов Spring в компоненты EJB.

• MVC на веб-уровне

• Поддержка электронной почты

• Поддержка планирования заданий

• Поддержка динамических сценариев (Groovy, JRuby, BeanShell)

# Spring Annotation

В начале был EJB 2.1, с его огромным количеством XML-файлов везде где только можно. Не будет особым преувеличением, если сказать, что на одну строку кода для бизнес-логики нужно было написать по крайней мере 10 строк кода от фреймворка и две страницы XML. Локальные и удалённые интерфейсы, ручной JNDI-lookup, многоуровневые try-catch, проверки на RemoteException. Потом создали [Spring framework](http://springframework.org/). Прошло время и Sun осознали урок. EJB 3.0 был даже проще Spring, XML-free, с аннотациями, dependency injection. 3.1 стал ещё одним огромным шагом в сторону упрощения.

По логике, EJB сейчас можно рассматривать как часть того, что предлагает Spring, почему нет реализации EJB в plain Spring, учитывая его поддержку из-коробки JPA 1.0/2.0, JSR-250, JSR-330, JAX-WS/RS и прочего. Spring framework сегодня воспринимается как медленный, тяжёлый и сложный для поддержки фреймворк, в-основном из-за XML-дескрипторов.

Уход от XML рассмотрим во втором примере Теоретической части.

Аннотации:

**@Component** - Аннотация для любого компонента фреймворка.

**@Service** - (Сервис-слой приложения) Аннотация, объявляющая, что этот класс представляет собой сервис – компонент сервис-слоя.

Сервис является подтипом класса @Component. Использование данной аннотации позволит искать бины-сервисы автоматически.

**@Repository** - (Доменный слой) Аннотация показывает, что класс функционирует как репозиторий и требует наличия прозрачной трансляции исключений. Преимуществом трансляции исключений является то, что слой сервиса будет иметь дело с общей иерархией исключений от Spring (DataAccessException) вне зависимости от используемых технологий доступа к данным в слое данных.

**@Controller** - (Слой представления) Аннотация для маркировки java класса, как класса контроллера. Данный класс представляет собой компонент, похожий на обычный сервлет (HttpServlet) (работающий с объектами HttpServletRequest и HttpServletResponse), но с расширенными возможностями от Spring Framework.

**@ResponseBody** - Аннотация показывает что данный метод может возвращать кастомный объект в виде xml, json...

**@RestController** - Аннотация аккумулирует поведение двух аннотаций @Controller и @ResponseBody

**@Transactional** - Перед исполнением метода помеченного данной аннотацией начинается транзакция, после выполнения метода транзакция коммитится, при выбрасывании RuntimeException откатывается.

**@Autowired** - Аннотация позволяет автоматически установить значение поля.

**@RequestMapping** - Аннотация используется для маппинга урл-адреса запроса на указанный метод или класс. Можно указывать конкретный HTTP-метод, который будет обрабатываться (GET/POST), передавать параметры запроса.

**@ModelAttribute** - Аннотация, связывающая параметр метода или возвращаемое значение метода с атрибутом модели, которая будет использоваться при выводе jsp-страницы.

**@PathVariable** - Аннотация, которая показывает, что параметр метода должен быть связан с переменной из урл-адреса.

**@Scope** - Аннотация для установки области жизни бина: *singleton* (только один экземпляр бина создается для IoC контейнера; значение по умолчанию ), *prototype* (создается новый экземпляр бина когда приходит запрос на его создание),*request* (один экземпляр бина для каждого HTTP запроса ), *session* (один экземпляр бина для каждой сессии), *globalSession* (один экземпляр бина для каждой глобальной сессии)

**@PostConstruct** - Аннотация для метода, который будет вызван после вызова конструктора бина.

**@PreDestroy** - Аннотация для метода, который будет вызван перед уничтожением бина.

**@Profile** - Аннотация для создания профилей конфигурации проекта. Может применяться как к бинам так и к конфигурационным классам.

# **Области видимости бинов**

**singleton** - Возвращает один и тот же экземпляр бина на каждый запрос контейнера Spring IoC (по умолчанию).

**prototype** - Создает и возвращает новый экземпляр бина на каждый запрос.

**request** - Создает и возвращает экземпляр бина на каждый HTTP запрос\*.

**session** - Создает и возвращает экземпляр бина для каждой HTTP сессии\*.

**global-session** - Создает и возвращает экземпляр бина для глобальной HTTP сессии\*.

Напомню, что сессия используется для получения физического соединения с базой данных (далее – БД). Благодаря тому, что сессия является легковесны объектом, его создают (открывают сессию) каждый раз, когда возникает необходимость, а потом, когда необходимо, уничтожают (закрывают сессию). Мы создаём, читаем, редактируем и удаляем объекты с помощью сессий.

Мы стараемся создавать сессии при необходимости, а затем уничтожать их из-за того, что ни не являются потоко-защищёнными и не должны юыть открыты в течение длительного времени.

Экземпляр класса может находиться в одном из трёх состояний:

* **transient**  
  Это новый экземпляр устойчивого класса, который не привязан к сессии и ещё не представлен в БД. Он не имеет значения, по которому может быть идентифицирован.
* **persistent**  
  Мы модем создать преходный экземпляр класса, связав его с сессией. Устойчивый экземпляр класса представлен в БД, а значение идентификатора связано с сессией.
* **detached**  
  После того, как сессия закрыта, экземпляр класса становится отдельным, независимым экземпляром класса.

В примитивном виде, транзакция выглядит примерно таким образом:

Ключевая функция Hibernate заключается в том, что мы можем взять занчения из нашего Java-класса и созранить их в таблице базы данных (далее – БД). С помощью конфигурационных файлов мы указываем Hibernate как извлечь данные из класса и соединить с определённым столбцами в таблице БД.

Если мы хотим, чтобы экземпляры (объекты) Java-класса в будущем созранялся в таблице БД, то мы называем их “сохранямые классы” (persistent class). Для того, чтобы сделать работу с Hibernate аксимально удобной и эффективной, мы следует использовать программную молель Простых Старых Java Объектов (Plain Old Java Object – POJO).

Сущесвуют определённые требования к POJO классам. Вот самые главные из них:

* Все классы должны иметь ID для простой идентификации наших объектов в БД и в Hibernate. Это поле класса соединяется с первичным ключём (primary key) таблицы БД.
* Все POJO – классы должны иметь конструктор по умолчанию (пустой).
* Все поля POJO – классов должны иметь модификатор доступа **private**иметь набор getter-ов и setter-ов в стиле JavaBean.
* POJO – классы не должны содержать бизнес-логику.

Мы называем классы POJO для того, чтобы подчеркнуть тот факт, что эти объекты являются экземплярами обычных Java-классов.

Во всех предыдущих примерах мы использовали конфигурационные XML – файлы для конфигурирования Hibernate. В этих XML – файлах мы указывали Hibernate с какой таблицей в нешей базе данных (далее – БД) необходимо связать тот или иной POJO – класс и к каким колонкам относятся те или иные поля в этом классе.

Но в Hibernate предусмотрена возможность конфигурирования прилоения с помощью аннотаций.

Аннотации являются мощным иструментом  для предоставления метаданных, а также намного нагляднее при чтении нашего кода другим разработчиком.

Обязательными аннотациями являются следующие:

**@Entity**

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

**@Table**

С помощью этой аннотации мы говорим Hibernate,  с какой именно таблицей необходимо связать (map) данный класс. Аннотация **@Table**имеет различные аттрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблец БД*.

**@Id**

С помощью аннотации **@Id**мы указываем *первичный ключ (Primary Key)*данного класса.

**@GeneratedValue**

Эта аннотация используется вместе с аннотацией **@Id** и определяет такие паметры, как **strategy** и **generator**.

**@Column**

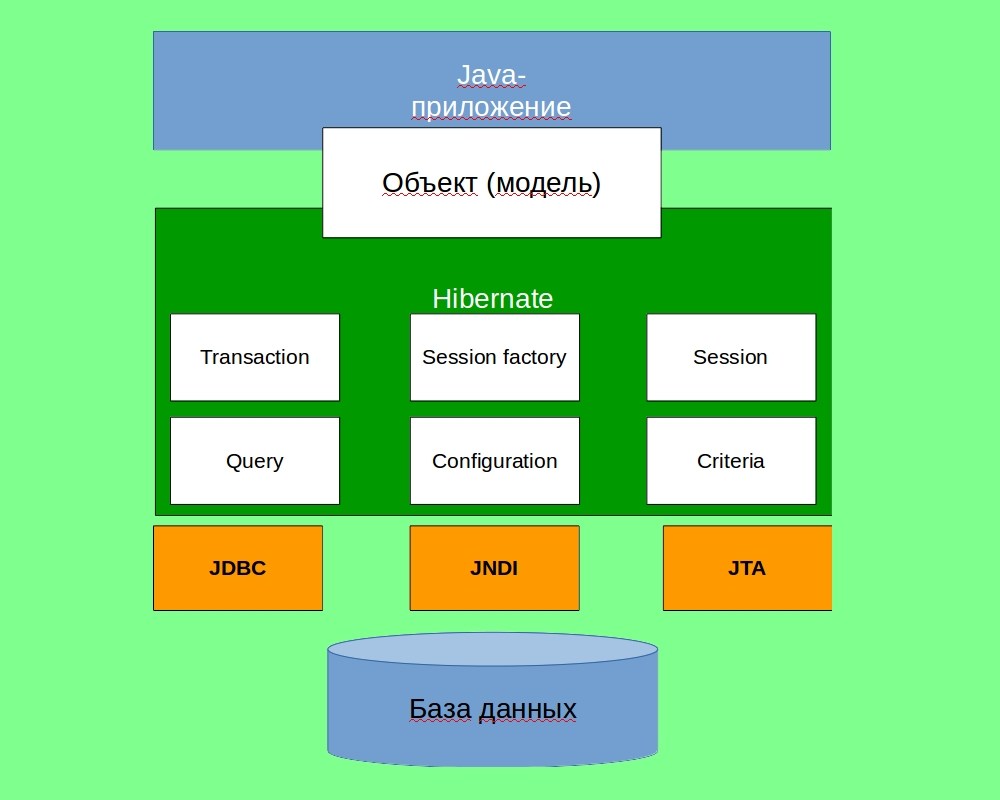
Аннотация **@Column**определяет к какому столбцу в таблице БД относится конкретное поле класса (аттрибут класса).

Наиболее часто используемые аттрибуты аннотации **@Column**такие:

* **name**Указывает имя столбца в таблице
* **unique**Определяет, должно ли быть данноезначение уникальным
* **nullable**Определяет, может ли данное поле быть NULL, или нет.
* **length**Указывает, какой размер столбца (например колчиство символов, при использовании String)

**Архитектура Hibernate**

Приложение, которое использует Hibernate (в крайне поверхностном представлении) имеет такую архитектуру:



Hibernate поддерживает такие API, как JDBC, JNDI, JTA.  
JDBC обеспечивает простейший уровень абстракции функциональности для реляционных БД. JTA и JNDI, в свою очередь, позволяют Hibernate использовать серверы приложений J2EE.

Давайье рассмотрим отдельно каждый из элементов Hibernate, которые указаны в схеме:

**Transaction**

Этот объект представляет собой рабочую единицу работы с БД. В Hibernate транзакции обрабатываются менеджером транзакций.

**SessionFactory**

Самый важный и самый тяжёлый объект (обычно создаётся в единственном эземпляре, при запуске прилодения). Нам необходима как минимум одна SessionFactory для каждой БД, каждый из которых конфигурируется отдельным конфигурационным файлом.

**Session**

Сессия исползуется для получения физического соединения с БД. Обычно, сессия создаётся при необходимости, а после этого закрывается. Это связано с тем, что эти объекты крайне легковесны. Чтобы понять, что это такое, модно сказать, что создание, чтение, изменение и удаление объектов происходит через объект Session.

**Query**

Этот объект использует HQL или SQL для чтения/записи данных из/в БД. Экземпляр запроса используется для связыания парметров запроса, ограничения количества результатов, которые будут возвращены и для выполнения запроса.

**Configuration**

Этот объект используется для создания объекта SessionFactory и конфигурирует сам Hibernate с помощью конифигурационного XML-файла, который объясняет, как обрабатывать объект Session.

**Criteria**

Используется для создания и выполнения объекто-ориентированных запроса дял получения объектов.

Практическая часть

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование поля | Тип поля | Обязательное (да, нет) | **Вариант N** |
| Фамилия | Текстовый | Да | Любой |
| Имя | Текстовый | Да | Любой |
| Отчество | Текстовый | Да | Любой |
| Дата рождения | Дата | Да | Любой |
| Пол | Boolean | Да | **Четный** |
| Серия паспорта | Текстовый | Да | Любой |
| № паспорта | Текстовый (с маской) | Да | Любой |
| Идент. номер | Текстовый (с маской) | Нет | **Четный** |
| Город проживания | **Список** (от 5 городов) | Да | Любой |
| Адрес факт.проживания | Текстовый | Да | Любой |
| Телефон дом | Текстовый (с маской) | Нет | **Четный** |
| Телефон моб | Текстовый (с маской) | Нет | Любой |
| E-mail | Текстовый | Нет | **Нечетный** |
| Трудостроен | Boolean | Нет | **Нечетный** |
| Должность | Текстовый | Нет | **Нечетный** |
| Город прописки | **Список** (от 5 городов) | Да | **1,4,7... (N%3==1)** |
| Адрес прописки | Текстовый | Да | **2,5,8... (N%3==2)** |
| Гражданство | **Список** | Да | Любой |
| Военнообязанный | Boolean | Да | **3,6,9... (N%3==0)** |

1. Выбрать поля базы данных для ввода и хранения информации о клиентах для выполнения лабораторной работы согласно **своему номеру варианта**.
2. Реализовать XML-файл с данными, его XSD-схему и API для добавления, чтения, обновления и удаления (CRUD) клиентов в СУБД из XML.
3. Разработать схему данных в которой для элементов типа **Список** предусмотрены отдельные справочники.
4. Реализовать схему данных в выбранной СУБД используя генератор классов **xjc** (**JAXB)** и **hibernate**.
5. На **хорошую** оценку дополнительно реализовать обратное преобразование (чтение из СУБД и маршаллинг в XML).
6. На **отличную** оценку предусмотреть аналогичное преобразование в JSON и обратно (рекомендуется gson).
7. На защите продемонстрировать операции преобразований из консоли, либо (рекомендуется) показать соответствующее покрытие тестами JUnit.