МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІГІВСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

JAVA ТЕХНОЛОГІЇ ПРИКЛАДНОГО ПРОГРАМУВАНН

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних, розрахунково-графічних робіт
та самостійної роботи здобувачів вищої освіти
з дисциплін
«Java та C# технології прикладного програмування»
та «Технології прикладного програмування»
для здобувачів вищої освіти
першого (бакалаврського) рівня вищої освіти
спеціальностей 121 – «Інженерія програмного забезпечення» та 123 – «Комп’ютерна інженерія»

Обговорено і рекомендовано
на засіданні кафедри ІТ та ПІ
протокол №11
від 15 червня 2020 р.

Чернігів ЧНТУ 2020

Java технології прикладного програмування. Методичні вказівки до виконання лабораторних, розрахунково-графічних робіт та самостійної роботи здобувачів вищої освіти з дисциплін «Java та C# технології прикладного програмування» та «Технології прикладного програмування» для здобувачів вищої освіти першого (бакалаврського) рівня вищої освіти спеціальностей 121 – «Інженерія програмного забезпечення» та 123 – «Комп'ютерна інженерія» / Укл.: Задорожній А.О., Богдан І. В., Войцеховська М.М., Дружинін О. О. – Чернігів: ЧНТУ, 2020. – 145 с.

Укладачі: Задорожній А. О., к.т.н., доцент
Богдан І. В., к.т.н.
Войцеховська М. М., асистент
Дружинін О. О., асистент

Рецензент: Дорош М. С., д.т.н., доцент

ЗМІСТ

СПИСОК УМОВНИХ СКОРОЧЕНЬ

СУРБД		Система Управління Реляційними Базами Даних
JDBC		Java Database Connectivity
ORM	-	Object Relational Mapping
DAO	-	Data Access Object
HTML	-	Hyper Text Markup Language
XML	-	Extensible Markup Language
SQL	-	Structured Query Language
CRUD	-	Create, Read, Update and Delete
MVC	-	Model View Controller
HQL		Hibernate Query Language

ВСТУП

Об’єктно-орієнтовані мови є ефективним інструментом для створення складних як настільних так і web-орієнтованих застосунків. З часу виникнення перших об’єктно-орієнтованих мов виник ряд підходів, які значно спрощують створення застосунків. Так, наприклад, однотипні дані прийнято зберігати в масивах або колекціях, для роботи з базами даних прийнято використовувати драйвери баз даних, написані під певну мову програмування. Досвід, накопичений методом спроб і помилок, поступово переріс в технології, що значно спрошують розробку застосунків.

Дані методичні вказівки створені, щоб допомогти отримати практичний досвід по роботі з базовим технологіями Java, технологіями взаємодії з базами даних, web-технологіями, технологіями інтеграції застосунків та технологіями створення web-сервісів так REST-сервісів.

В даних методичних вказівках розглядаються такі технології, як Hibernate – технологія для об’єктно-реляційного відображення, Servlet і JSP – технології створення web-застосунків, Apache CXF – технологія для створення web-сервісів, Jersey – технологія для створення REST-сервісів, а також розглядаються деякі аспекти використання базових технологій платформи Java, таких як рефлексія.

Деякі технології, описані в даних методичних вказівках, використовуються не тільки в платформі Java, але і платформі .NET. Наприклад, технологія Hibernate використовується в .NET під назвою NHibernate. Таким чином, дані методичні вказівки повинні допомогти сформувати загальну картину не тільки для платформи Java але і для платформи .NET.

В деяких лабораторних роботах описані шаблони проектування, які доцільно застосовувати при вирішенні тої чи іншої задачі. Наприклад, в лабораторній роботі з об’єктно-реляційного відображення розглядається архітектурний шаблон Data Access Object, а також шаблон Factory. Це повинно допомогти сформувати уявлення про використання архітектурних шаблонів і шаблонів проектування при створенні застосунків з використання об’єктно-орієнтованих мов програмування.

1 ЛАБОРАТОРНА РОБОТА №1
ВИВЧЕННЯ ДИНАМІЧНОГО ВИЗНАЧЕННЯ ТИПІВ

1.1 Мета роботи

Вивчити механізм динамічного визначення типів, навчитися застосовувати механізм динамічного завантаження довільного класу по заданому імені.

1.2 Теоретичні відомості

1.2.1 Клас Class і способи отримання його екземпляру

В Java вся інформація про клас зберігається в спеціальному об’єкті типу Class. Для кожного класу в програмі існує свій об’єкт Class, який створюється при завантаженні класу, створеного на етапі компіляції і збереження на диску в файлі з розширенням class.

Існує декілька способів як отримати екземпляр класу Class, що відповідає даному класу чи інтерфейсу:

1. Використати літерал об’єкту Class, додавши до імені класу суфікс *.class*, наприклад, *Class clazz = String.class*. Літерали об’єктів Class працюють з регулярними класами, а також з інтерфейсами, масивами і примітивними типами. *Class clazz = byte[].class*.
2. Якщо ми маємо екземпляр деякого класу, можна викликати метод *getClass()*, що присутній в будь-якому Java-об’єкті (успадкованому від класу Object).
3. Викликати статичний метод класу *Class.forName()*, який повертає екземпляр *Class: Class clazz = Class.forName("Повне ім’я класу")*.
4. Додатково існує стандартне поле з назвою TYPE, яке існує для кожного класу обгортки примітивного типу. Поле TYPE створює посилання на об’єкт класу для відповідного примітивного класу, наприклад, *Class clazz = Byte.TYPE*.

1.2.2 Динамічне завантаження і створення екземплярів класів

В Java вбудована можливість динамічного завантаження довільного класу по заданому імені, яка реалізована в класі Class.

```
public static Class forName(String className) throws ClassNotFoundException;
```

Метод відшукує в системі (в шляху пошуку CLASSPATH) клас з заданим іменем className і повертає відповідний екземпляр класу Class. Ім’я класу повинне бути повним, тобто, включати ім’я пакету. Якщо такий клас відсутній, створюється виключення ClassNotFoundException. Після того як отриманий об’єкт типу Class можна створити екземпляр цього класу.

```
Class clazz = Class.forName(String className);
Object object = clazz.newInstance();
```

Для успішного виконання цього методу у класі повинен бути описаний конструктор без параметрів (конструктор по за замовчуванням), інакше буде створено виключну ситуацію InstantiationException.

Щоб працювати з таким об'єктом необхідно знати методи класу, їх параметри і значення, що повертається. Для того, щоб вказати контракт класу, використовують інтерфейси.

Розглянемо приклад. Необхідно реалізувати застосунок з можливістю перекладу тексту з однієї мови на іншу.

```
public interface LanguageTranslator{
  /** Перекладає текст src з мови srcLang на мову dstLang
  */
  public String translate(String src, String srcLang, String dstLang);
}
```

Нехай програма не вміє виконувати переклад, але її можна розширити класом сторонніх розробників, які цю задачу вирішувати вміють. Всі ці класи повинні реалізовувати інтерфейс LanguageTranslator.

```
String source="текст, що потребує перекладу";
String srcLang="UR";
String dstLang="EN";
Class clazz = Class.forName("Повне ім'я класу перекладача")
Object o = clazz.new Instance();
if (!(o instanceof LanguageTranslator)){
  throw new Exception("Вказаний клас не реалізує інтерфейс перекладача");
}
String result = ((LanguageTranslate) o).translate(source, srcLang, dstLang);
```

На описаній техніці можуть будуватись системи плагінів – модулів, що додаються до системи, що вже працює.

1.2.3 Дослідження класів

В класі Class<T> існує спеціальний набір методів для дослідження класів і відповідні методи для перевірки типу класу. Можна отримати список всіх конструкторів, методів, полів, анотацій, описаних в класі.

Таблиця 1.1 – Методи для дослідження класів

1	2
Annotation, Annotation[]	getAnnotatin(), getAnnotations(), getDeclaredAnnotation(), getDeclaredAnnotations()
Cinstructor<T>, Constructor[]	getConstructro(Class …param), getConctrunctors(), getDeclaredConstructor(), getDeclaredConstructors()

1	2
Method, Method[]	getMethod(String name, Class … param), getMethods(), getDeclaredMethod(String name, Class … param), getDeclaredMethods()
Field, Field[]	getField(String name), getFields(), getDeclaredField(String name), getDeclaredFields()
Class[]	getInterfaces()
Class[]	getClasses(), getDeclaredClasses()
T[]	getEnumConstatnts()
Class<? super T>	getSuperclass()

Таблиця 1.2 – Методи для перевірки типу класу

1	2
boolean isAnnotation()	Returns true if this Class object represents an annotation type.
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)	Returns true if an annotation for the specified type is present on this element, else false.
boolean isAnonymousClass()	Returns true if and only if the underlying class is an anonymous class.
boolean isArray()	Determines if this Class object represents an array class.
boolean isAssignableFrom(Class<?> cls)	Determines if the class or interface represented by this Class object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified Class parameter.
boolean isEnum()	Returns true if and only if this class was declared as an enum in the source code.
boolean isInstance(Object obj)	Determines if the specified Object is assignment-compatible with the object represented by this Class.
boolean isInterface()	Determines if the specified Class object

1	2
	represents an interface type.
boolean isLocalClass()	Returns true if and only if the underlying class is a local class.
boolean isMemberClass()	Returns true if and only if the underlying class is a member class.
boolean isPrimitive()	Determines if the specified Class object represents a primitive type.

1.2.4 Розкриття модифікаторів класу

Для розкриття модифікаторів отриманих членів класу використовується метод int getModifiers(), який повертає модифікатор у вигляді цілого. Всі можливі модифікатори мови визначені в класі java.lang.reflect.Modifier:

PRIVATE, PROTECTED, PUBLIC
ABSTRACT, FINAL, STATIC
INTERFACE
SYNCHRONIZED
TRANSIENT, VOLATILE
NATIVE

Для кожного з модифікаторів визначений відповідний метод boolean isXXXX(int m).

```
public Class SampleModifier {
  public static void main(String[] args){
    String s = new String();
    printModifiers(s);
  }
  public static void printModifiers(Object o){
    Class c = o.getClass();
    int m = c.getModifiers();
    if (Modifier.isPublic(m)) System.out.println("public");
    if (Modifier.isAbstract(m)) System.out.println("abstract");
    if (Modifier.isFinal(m)) System.out.println("final");
  }
}
```

1.2.5 Динамічний виклик методів

Окрім дослідження структури класів механізм рефлексії забезпечує можливість динамічного виклику методів під час виконання програми, що найчастіше використовується для реалізації механізму розширень (плагінів), які дозволяють додавати до базового функціонал програми додавати нові можливості.

Розглянемо, як можна здійснити динамічний виклик методів, що розмішені в класі, представленому нижче:

```
public class Operations {
  public double publicSum(int a, double b) {
    return a + b;
  }
  public static double publicStaticMultiply(float a, long b) {
    return a * b;
  }
  private boolean privateAnd(boolean a, boolean b) {
    return a && b;
  }
  protected int protectedMax(int a, int b) {
    return a > b ? a : b;
  }
}
```

Для того, щоб динамічно викликати метод перш за все необхідно отримати посилання на цей метод. Нижче приведені приклади, як отримати посилання на методи, описані в класі Operations:

```
//Використовують, якщо метод публічний
Method sumInstanceMethod = Operations.class.getMethod("publicSum", int.class,
double.class);
Method multiplyStaticMethod =
Operations.class.getMethod("publicStaticMultiply", float.class, long.class);

//Використовують, якщо метод публічний, приватний, або захищений
Method andPrivateMethod = Operations.class.getDeclaredMethod("privateAnd",
boolean.class, boolean.class);
Method maxProtectedMethod = Operations.class.getDeclaredMethod("protectedMax",
int.class, int.class);
```

Приклад виклику звичайного (не статичного) методу, приведений нижче:

```
Method sumInstanceMethod = Operations.class.getMethod("publicSum", int.class,
double.class);
Operations operationsInstance = new Operations();
Double result = (Double) sumInstanceMethod.invoke(operationsInstance, 1, 3);
```

Приклад виклику статичного методу приведений нижче:

```
Method multiplyStaticMethod =
Operations.class.getDeclaredMethod("publicStaticMultiply", float.class,
long.class);
Double result = (Double) multiplyStaticMethod.invoke(null, 3.5f, 2);
```

За замовчуванням не всі методи доступні для динамічного виклику з допомогою механізму рефлексії. Це значить що JVM виконує перевірку доступності методу під час його виклику. Наприклад, захищений метод буде неможливо виконати без додаткового виклику методу setAccessible. Приклад виклику захищеного методу приведений нижче:

```
Method andPrivateMethod = Operations.class.getDeclaredMethod("privateAnd",
boolean.class, boolean.class);
andPrivateMethod.setAccessible(true);
Operations operationsInstance = new Operations();
Boolean result = (Boolean) andPrivateMethod.invoke(operationsInstance, true,
false);
```

1.2.6 Завантажувачі класів

При першому зверненні до класу віртуальна машина робить спробу завантажити цей клас в пам’ять. Стандартному завантажувачу класів передається повне ім’я класу для пошуку і завантаження. Стандартний завантажувач виконує пошук класів в каталогах і архівах, вказаних в змінній середовища CLASSPATH. Якщо виникає необхідність завантажити клас з довільного місця (не вказаного в CLASSPATH), то необхідно написати свій власний завантажувач класів, який розширює клас ClassLoader.

Нижче представлений текст програми власного завантажувача класів.

```
import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class MyClassLoader extends ClassLoader {
  private String jarName;
  private JarFile jar;
  private Map<String, Class<?>> loaded = new HashMap<String, Class<?>>();

  public MyClassLoader(String jarName) {
    super(MyClassLoader.class.getClassLoader());
    this.jarName = jarName;
    try {
      this.jar = new JarFile(jarName);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public Class<?> findClass(String name) throws ClassNotFoundException {
    Class<?> clazz = loaded.get(name);
    if (clazz != null)
      return clazz;
    try {
      return findSystemClass(name);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    byte[] data;
    try {
      data = loadClassData(name);
      clazz = defineClass(name, data, 0, data.length);
      loaded.put(name, clazz);
    } catch (Throwable ex) {
      throw new ClassNotFoundException(ex.getMessage());
    }
```

```
    return clazz;
  }

  private byte[] loadClassData(String name) throws ClassNotFoundException {
    String entryName = name.replace('.', '/') + ".class";
    byte buf[] = new byte[0];
    try {
      JarEntry entry = jar.getJarEntry(entryName);
      if (entry == null) {
        throw new ClassNotFoundException(name);
      }
      InputStream input = jar.getInputStream(entry);
      int size = new Long(entry.getSize()).intValue();
      buf = new byte[size];
      int count = input.read(buf);
      if (count < size)
        throw new ClassNotFoundException("Error reading class " +
          name + " from :" + jarName);
    } catch (IOException ex) {
      throw new ClassNotFoundException(ex.getMessage());
    }
    return buf;
  }

}
```

Нижче представлений текст класу TestClass, що демонструє роботу власного завантажувача класу.

```
package Labs;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Enumeration;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
import java.lang.ClassLoader;
public class TestClass{
  public static void main(String[] args) throws FileNotFoundException,
    IOException, ClassNotFoundException {
    ClassLoader loader = new MyClassLoader(args[0]);
    JarFile f = new  JarFile(args[0]);
    Enumeration<JarEntry> enumiration = f.entries();
    String s;
    Class c;
    JarEntry e;
    while(enumiration.hasMoreElements()){
      e = enumiration.nextElement();
      if (e.getName().contains(".class")) {
        System.out.print(e.getName()+":");
        s = e.getName().replaceAll("/", ".");
        s = s.substring(0, s.length()-6);
```

```
        try{
          c = Class.forName(s, false, loader);
          if (c!=null){
            System.out.println(":"+ClassDetail.getInfo(c));
          }
        } catch(Exception e2){
          System.out.println("Error:"+e2.getMessage());
        }
      }else{
        System.out.println(e.getName());
      }
    }
    f.close();
  }
}
```

1.2.7 Анотації користувача

Анотації являють собою дескриптори, що включені в текст програми, і використовуються для зберігання метаданих програмного коду, необхідних на різних етапах життєвого циклу програми. Інформація, що зберігається в анотаціях, може використовуватися відповідними обробниками для створення необхідних допоміжних файлів для маркування класів, полів і т. д.

Анотація задається описом відповідного інтерфейсу. Наприклад:

```
import java.lang.annotation.*;
@Target(value=ElementType.FIELD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface Name{
  String name();
  String type() default "string";
}
```

Як видно з прикладу, анотація визначається описом ключового слова @interface і може включати в себе декілька полів, які можна задати як обов’язковими, так і не обов’язковими. В останньому випадку надається default значення поля. Також з прикладу видно, що в саму анотацію можна розмістити декілька анотацій. Анотація @Retention дозволяє вказати життєвий цикл анотації: буде вона присутньою тільки в вихідному коді, в скомпільованому файлі, або її буде видно під час виконання програми. Вибір необхідного типу залежить від того, як ви хочете використовувати анотацію, наприклад, генерувати щось побічне з вихідного коду або в процесі виконання звертатися до класу через reflection. Анотація @Target вказує, що саме ми можемо помітити даною анотацією. Це може бути поле, метод, або тип. Анотація @Document вказує, що помічена таким чином анотація повинна бути додана в javadoc поля або метода. Анотація @Inherit помічає анотацію, яка буде успадкована, потомком класу, відміченого такою анотацією.

Розглянемо приклад використання анотацій. Нехай у нас є застосунок, який на вхід отримує клас помічений анотаціями користувача, для того, щоб застосунок міг управляти життєвим циклом об’єктів цього класу і нехай там

будуть анотації StartObject і StopObject для опису методів класу і ControlledObject для опису самого класу. Анотація ControlledObject буду мати поле name для зберігання імені класу. Анотації будуть виглядати наступним чином:

```
@Target(value=ElementType.METHOD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface StartObject{
}

@Target(value=ElementType.METHOD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface StopObject{
}

@Target(value=ElementType.TYPE)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface ControlledObject{
  String name();
}
```

Код, помічений анотаціями, буде виглядати наступним чином:

```
@ControlledObject(name="mycontrolledobject")
public class Cookies{
  @StartObject
  public void createCookie(){
    //бізнес логіка
  }
  @StopObject
  public void stopCookie(){
    //бізнес логіка
  }
}
```

Код застосунку, що буде перевіряти, чи підходить клас під наші вимоги приведений нижче:

```
Class cl = Class.forName("ua.cn.stu.Cookies");
if(!cl.isAnnotationPresent(ControlledObject.class)){
  System.err.println("no annotation");
} else {
  System.out.println("name - " + cl.getAnnotation(ControlledObject.class));
}
boolean hasStart=false;
boolean hasStop=false;
Method[] method = cl.getMethods();
for(Method md: method){
  if(md.isAnnotationPresent(StartObject.class)) {hasStart=true;}
  if(md.isAnnotationPresent(StopObject.class)) {hasStop=true;}
  if(hasStart && hasStop){break;}
}
System.out.println("Start annotaton  - " + hasStart + ";  Stop annotation  - "
+ hasStop );
```

1.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення калькулятора, побудованого на механізмі плагінів. Калькулятор буде складатися з ядра, основною функцією якого є завантаження плагінів, і з декількох плагінів, які розширюють калькулятор декількома бінарними арифметичними операціями. Процес створення калькулятора буде відбуватися з використанням середовища Eclipse.

1. Відкрийте Eclipse і створіть новий Java проект.
2. В каталозі src створіть пакет *ua.cn.stu.plugin.api*, що буде містити інтерфейси, необхідні для взаємодії з плагінами.
3. Додайте до створеного пакету наступні інтерфейси:

```
public interface Plugin {
  void invoke();
}

public interface UnaryOperator {
  double calculateUnary(double operand);
}

public interface BinaryOperator {
  double calculateBinary(double firstOperand, double secondOpearnd);
}
```

Інтерфейс *Plugin* використовується для перевірки, чи є завантажений клас плагіном калькулятора чи ні. Інтерфейси *UnaryOperator* і *BinaryOperator* використовуються для визначення API взаємодії ядра калькулятора з плагінами.

4. Створіть каталог plugins, який буде містити jar-архіви плагінів калькулятора.
5. Створіть пакет *ua.cn.stu.core*, в якому будуть знаходитися базові класи калькулятора.
6. В пакеті *ua.cn.stu.core* створіть клас *CoreCalculator* з *main* методом:

```
public class CoreCalculator {

  public static void main(String[] args ){
  }
}
```

7. Додайте до пакету *ua.cn.stu.core* перерахування для зберігання типу операції:

```
public enum OperatorType {
  BINARY("binary"), UNARY("unary");
  private String operatorTypeString;

  OperatorType(String operatorTypeString) {
    this.operatorTypeString = operatorTypeString;
  }

  public String getOperatorType() {
    return operatorTypeString;
```

```
  }
}
```

8. Додайте до пакету *ua.cn.stu.core* клас для зберігання даних про завантажені класи

```
public class PluginInfo {

  private Class<?> classReference;
  private OperatorType operatorType;
  private String operator;
  private String description;

  public Class<?> getClassReference() {
    return classReference;
  }

  public void setClassReference(Class<?> classReference) {
    this.classReference = classReference;
  }

  public OperatorType getOperatorType() {
    return operatorType;
  }

  public void setOperatorType(OperatorType operatorType) {
    this.operatorType = operatorType;
  }

  public String getOperator() {
    return operator;
  }

  public void setOperator(String operator) {
    this.operator = operator;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }
}
```

9. Додайте до пакету *ua.cn.stu.core* клас-завантажувач для завантаження плагінів з jar-архівів.

```
public class PluginClassLoader extends ClassLoader {
  private String jarName;
  private JarFile jar;
  private Map<String, Class<?>> loaded = new HashMap<String, Class<?>>();
```

```
public PluginClassLoader(String jarName) {
  super(PluginClassLoader.class.getClassLoader());
  this.jarName = jarName;
  try {
    this.jar = new JarFile(jarName);
  } catch (IOException ex) {
    ex.printStackTrace();
  }
}

public Class<?> findClass(String name) throws ClassNotFoundException {
  Class<?> clazz = loaded.get(name);
  if (clazz != null)
    return clazz;
  try {
    return findSystemClass(name);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  byte[] data;
  try {
    data = loadClassData(name);
    clazz = defineClass(name, data, 0, data.length);
    loaded.put(name, clazz);
  } catch (Throwable ex) {
    throw new ClassNotFoundException(ex.getMessage());
  }
  return clazz;
}

private byte[] loadClassData(String name) throws ClassNotFoundException {
  String entryName = name.replace('.', '/') + ".class";
  byte buf[] = new byte[0];
  try {
    JarEntry entry = jar.getJarEntry(entryName);
    if (entry == null) {
      throw new ClassNotFoundException(name);
    }
    InputStream input = jar.getInputStream(entry);
    int size = new Long(entry.getSize()).intValue();
    buf = new byte[size];
    int count = input.read(buf);
    if (count < size)
      throw new ClassNotFoundException("Error reading class " + name +
        " from :" + jarName);
  } catch (IOException ex) {
    throw new ClassNotFoundException(ex.getMessage());
  }
  return buf;
```

```
  }
}
```

10. Створіть пакет *ua.cn.stu.plugin.plusoperation* і додайте до нього клас плагіна, що вміє додавати два числа:

```
public class PlusPlugin implements Plugin, BinaryOperator {

  public static final String PLUGIN_NAME = "Plus operation plugin";

  @Override
  public void invoke() {
    System.out.println(PLUGIN_NAME + "loaded");
  }

  @Override
  public double calculateBinary(double firstOperand, double secondOpearnd) {
    return firstOperand + secondOpearnd;
  }
}
```

11. Додайте до пакету *ua.cn.stu.plugin.plusoperation* propery-файл дескриптор плагіну:

```
main.class=ua.cn.stu.plugin.plusoperation.PlusPlugin
type=binary
operator=+
description=Plus operation
```

12. Створіть пакет *ua.cn.stu.plugin.minusoperation* і додайте до нього клас плагіна, що вміє віднімати два числа:

```
public class MinusPlugin implements Plugin, BinaryOperator {

  public static final String PLUGIN_NAME = "Minus operation plugin";
  @Override
  public void invoke() {
    System.out.println(PLUGIN_NAME + " loaded");
  }

  @Override
  public double calculateBinary(double firstOperand, double secondOpearnd) {
    return firstOperand - secondOpearnd;
  }
}
```

13. Додайте до пакету *ua.cn.stu.plugin.minusoperation* property-файл дескриптор плагіну:

```
main.class=ua.cn.stu.plugin.minusoperation.MinusPlugin
type=binary
operator=-
description=Minus operation
```

14. Створіть jar-архіви плагінів додавання і віднімання двох чисел. В головному меню Eclipse натисніть File->Export, оберіть JAR file і натисніть Next. Кожен jar-архів плагіна моє містити всі класи пакету *ua.cn.stu.plugin.api*,

клас плагіна і property-файл дескриптора плагіна. На рисунку 1.1 зображено, які ресурси повинні бути присутні в jar-архіві плагіна додавання двох чисел. Оберіть опцію Export generated class files and resources, оберіть каталог для збереження плагіна і натисніть Finish. Всі плагіни в рамках даної лабораторної роботи повинні бути збережені в каталог plugins проекту. Процес створення плагіну для віднімання двох чисел є подібним до процесу створення плагіну для додавання двох чисел.

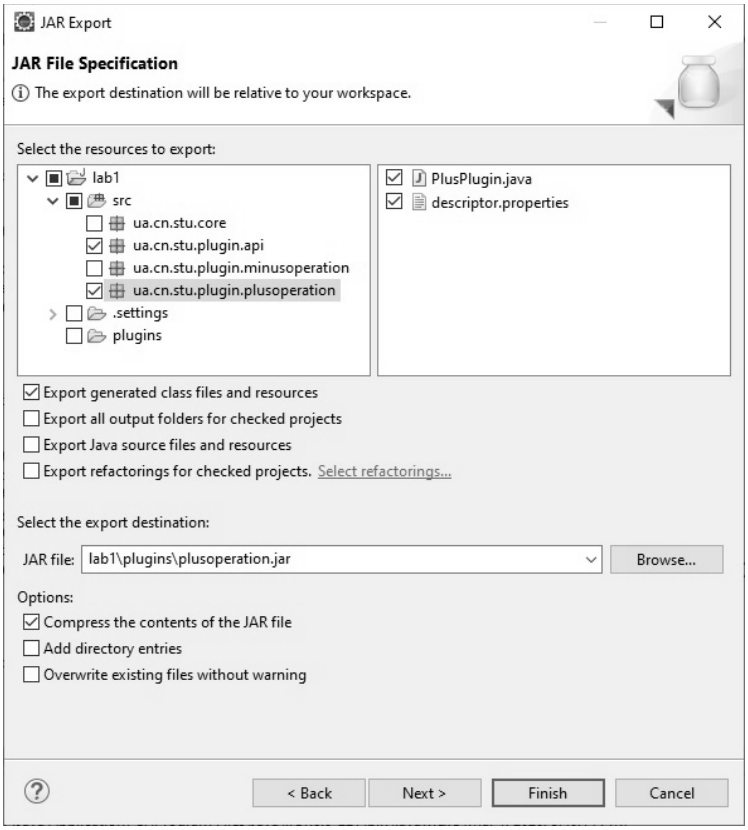

Рисунок 1.1 – Ресурси плагіна додавання двох чисел

15. Додайте до класу *CoreCalculator* набір констант, що будуть використовуватися в різних методах даного класу:

```
private static final String DESCRIPTION_PROPERTY = "description";
private static final String DESCRIPTOR_NAME_PART = "descriptor";
private static final String OPERATOR_PROPERTY = "operator";
private static final String TYPE_PROPERTY = "type";
private static final String MAIN_CLASS_PROPERTY = "main.class";
private static final String PROPERTIES_EXTENTION = ".properties";
private static final String JAR_EXTENTION = ".jar";
private static final String PLUGIN_DIR = "plugins";
```

16. Додайте до класу *CoreCalculator* метод для отримання всіх jar-файлів з каталогу plugins:

```
private static File[] getAllJarsFromPluginDir() {
  File pluginDir = new File(PLUGIN_DIR);
  File[] jars = pluginDir.listFiles(new FileFilter() {
    @Override
    public boolean accept(File pathname) {
      return pathname.isFile() && pathname.getName().endsWith(JAR_EXTENTION);
```

```
    }
  });
  return jars;
}
```

17. Додайте до класу *CoreCalculator* метод для динамічного виклику методів за іменем з класу:

```
private static Object executeMethod(Class<?> pluginClass, String methodName,
Class<?>[] methodParameterTypes, Object[] methodArguments) throws
NoSuchMethodException, IllegalAccessException, InvocationTargetException,
InstantiationException {
  Method method = pluginClass.getMethod(methodName, methodParameterTypes);
  Object result = method.invoke(pluginClass.newInstance(), methodArguments);
  return result;
}
```

18. Додайте до класу *CoreCalculator* метод для перевірки, чи є завантажений клас класом-плагіном калькулятора:

```
private static boolean isPluginClass(Class<?> pluginClass) {
  boolean isPlugin = false;
  Class<?>[] implementedInterfaces = pluginClass.getInterfaces();
  for (Class<?> implementedInterface : implementedInterfaces) {
    if ("ua.cn.stu.plugin.api.Plugin"
      .equalsIgnoreCase(implementedInterface.getName())) {
      isPlugin = true;
      continue;
    }
  }
  return isPlugin;
}
```

19. Додайте до класу *CoreCalculator* метод для визначення місця знаходження property-файлу дескриптора плагіна в jar-архіві:

```
private static String getDescriptorPath(URL jarURL) throws IOException {
  String descpriptorPath = null;
  ZipInputStream zip = new ZipInputStream(jarURL.openStream());
  ZipEntry zipEntry = null;
  while ((zipEntry = zip.getNextEntry()) != null) {
    String entryName = zipEntry.getName();
    if (entryName.contains(DESCRIPTOR_NAME_PART) &&
      entryName.endsWith(PROPERTIES_EXTENTION)) {
      descpriptorPath = entryName;
      continue;
    }
  }
  return descpriptorPath;
}
```

20. Додайте до класу *CoreCalculator* метод для завантаження класів-плагінів:

```
private static Map<String, PluginInfo> loadPlugins(File[] jars) throws
URISyntaxException, NoSuchMethodException, InvocationTargetException {
  Map<String, PluginInfo> pluginClasses = new HashMap<String, PluginInfo>();
```

```
  for (File jar : jars) {
    PluginClassLoader pluginClassloader = null;
    try {
      URL jarURL = jar.toURI().toURL();
      pluginClassloader = new PluginClassLoader(jar.getPath());
      String descpriptorPath = getDescriptorPath(jarURL);
      Properties propertis = new Properties();
      InputStream inputStream = pluginClassloader
        .getResourceAsStream(descpriptorPath);
      propertis.load(inputStream);
      String className = propertis.getProperty(MAIN_CLASS_PROPERTY);
      String operationType = propertis.getProperty(TYPE_PROPERTY);
      String operator = propertis.getProperty(OPERATOR_PROPERTY);
      String description = propertis.getProperty(DESCRIPTION_PROPERTY);
      Class<?> pluginClass = Class
        .forName(className, false, pluginClassloader);
      boolean isPlugin = isPluginClass(pluginClass);
      if (isPlugin && operationType != null && operator != null
        && description != null) {
        executeMethod(pluginClass, "invoke", new Class<?>[] {},
          new Object[] {});
        PluginInfo pluginInfo = new PluginInfo();
        pluginInfo.setClassReference(pluginClass);
        if (OperatorType.UNARY.getOperatorType()
          .equalsIgnoreCase(operationType)) {
          pluginInfo.setOperatorType(OperatorType.UNARY);
        } else if (OperatorType.BINARY.getOperatorType()
          .equalsIgnoreCase(operationType)) {
            pluginInfo.setOperatorType(OperatorType.BINARY);
        }
        pluginInfo.setOperatorType(OperatorType.BINARY);
        pluginInfo.setOperator(operator);
        pluginInfo.setDescription(description);
        pluginClasses.put(pluginInfo.getOperator().toString(), pluginInfo);
      }
    } catch (MalformedURLException ex) {
      ex.printStackTrace();
    } catch (ClassNotFoundException ex) {
      ex.printStackTrace();
    } catch (IOException ex) {
      ex.printStackTrace();
    } catch (InstantiationException ex) {
      ex.printStackTrace();
    } catch (IllegalAccessException ex) {
      ex.printStackTrace();
    }
  }
  return pluginClasses;
}
```

21. Змініть головний метод класу *CoreCalculator* наступним чином:

```
public static void main(String[] args) throws IOException,
InstantiationException, IllegalAccessException, URISyntaxException,
NoSuchMethodException, SecurityException, IllegalArgumentException,
InvocationTargetException {
  File[] jars = getAllJarsFromPluginDir();
  Map<String, PluginInfo> pluginClasses = loadPlugins(jars);
  System.out.println("Supported operations:");
  for (PluginInfo pluginInfo : pluginClasses.values()) {
    System.out.println(pluginInfo.getDescription());
  }
  String input = null;
  while (!"exit".equalsIgnoreCase(input)) {
    System.out.println("Please enter expression or type exit >>>");
    BufferedReader reader = new BufferedReader(
      new InputStreamReader(System.in));
    input = reader.readLine();
    if ("exit".equalsIgnoreCase(input)) {
      System.out.println("Closing calculator");
    } else {
      boolean isMatchToOperation = false;
      for (String operation : pluginClasses.keySet()) {
        if (input.contains(operation)) {
          Pattern pattern = Pattern.compile(
            "(\\d+)(\\" + operation + ")(\\d+)");
          Matcher matcher = pattern.matcher(input);
          if (matcher.matches()) {
            isMatchToOperation = true;
            String firstParameter = matcher.group(1);
            String opearator = matcher.group(2);
            String secondParameter = matcher.group(3);
            Class<?>[] methodParameterTypes = new Class<?>[] {
              double.class, double.class };
            Object[] methodArgument = new Object[] {
              new Double(firstParameter), new Double(secondParameter) };
            Double result = (Double) executeMethod(
              pluginClasses.get(opearator).getClassReference(),
              "calculateBinary", methodParameterTypes, methodArgument);
            System.out.println("The result of opereation " + result);
          }
        }
      }
      if (!isMatchToOperation) {
        System.out.println("Operatio is not supported");
      }
    }
  }
}
```

22. Запустіть клас *CoreCalculator* на виконання і перевірте працездатність програми.

23. Вдоскональте застосунок відповідно до завдання на лабораторну роботу.

1.4 Завдання на лабораторну роботу

Реалізувати пагін для описаного в лабораторній роботі калькулятора відповідно до варіанту завдання з таблиці 1.3. Зверніть увагу, що в лабораторній роботі описаний процес створення плагіна для бінарної операції, а в завданні до лабораторної роботи треба створити плагін для унарної операції.

Таблиця 1.3 – Варіанти завдань

Остання цифра залікової книжки	Функція калькулятора	Приклад виклику
0	Корінь квадратний (sqrt)	sqrt x
1	Возведення в квадрат (sqr)	sqr x
2	Синус (sin)	sin x
3	Косинус (cos)	cos x
4	Тангенс (tg)	tg x
5	Котангенс (ctg)	ctg x
6	Факторіал (!)	x!
7	Натуральний логарифм (ln)	ln x
8	Арксинус (arcsin)	arcsin x
9	Арккосинус (arccos)	arccos x

1.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

1.6 Контрольні питання

1. Яке призначення класу Class;
2. Як отримати екземпляр класу Class, що відповідає даному класу;
3. Поясніть визначення конструктора за замовчуванням;
4. Який механізм отримання RTTI;
5. Як здійснити завантаження довільного класу за іменем;
6. Назвіть основні класи пакету java.lang.reflect в Java;
7. Яким чином здійснюється розкриття модифікаторів класу;
8. Назвіть методи для перевірки типу класу;
9. Reflection і безпека;
10. Для чого використовуються анотації в Java.

1.7 Завдання до захисту лабораторної роботи

1. Додати до калькулятора плагін, що вміє складати три числа.
2. Додати до калькулятора плагін, що вміє перетворювати фунти в кілограми.
3. Додати до калькулятора плагін, що вміє перетворювати фути в метри.
4. Додати до калькулятора плагін, що вираховує корінь будь-якого ступеня.
5. Додати до калькулятора плагін, що вираховує логарифм будь-якого ступеня.
6. Додати до калькулятора плагін, що знаходить кількість входжень заданого символу в рядок.
7. Додати до калькулятора плагін, що видаляє з рядка заданий підрядок.
8. Додати до калькулятору плагін, що сортує масив чисел введений через крапку з комою.
9. Замінити property-файли дескрипторів плагінів на рівнозначні анотації.
10. Створити програму, виводить на екран дерево класів у наступному вигляді: A1(A2, A3 (A4, A5)), B1 (B2, B3), де A2, A3 – наслідники класу A1, A4, A5 – наслідники класу A3, B2, B3 – наслідники класу B1.

2 ЛАБОРАТОРНА РОБОТА №2
ДОСТУП ДО СУБД З ВИКОРИСТАННЯМ ПАКЕТУ JDBC

2.1 Мета роботи

Вивчити основні прийоми підключення до реляційних баз даних з Java застосунку, навчитися виконувати запити на отримання даних з бази даних, а також виконувати запити на додання, зміну і видалення даних.

2.2 Теоретичні відомості

2.2.1 Вступ до JDBC

Розробниками компанії Sun був створений спеціалізований пакет java.sql, який дозволяв програмі, написаній на мові Java здійснювати доступ до системи управління реляційними базами даних (СУРБД, в подальшому просто БД). Завдяки пакету JDBC (Java Database Connectivity) можна підключатися до реляційної бази даних і взаємодіяти з нею, використовуючи мову управління базою даних – SQL. Архітектура JDBC приведена на рисунку 2.1.

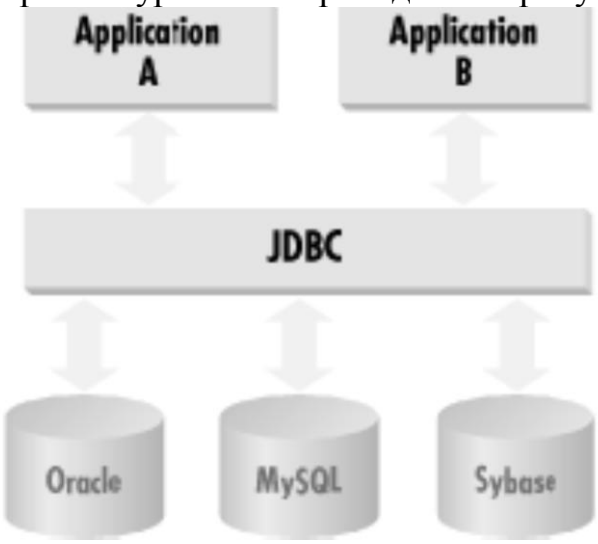

Рисунок 2.1 – Архітектура JDBC

JDBC є протоколом, а не інтерфейсом, і побудований на специфікаціях SAG CLI (SQL Access Group Call Level Interface - інтерфейс рівня виклику групи доступу SQL). Існує 4 різних види драйверів (рисунок 2.2).

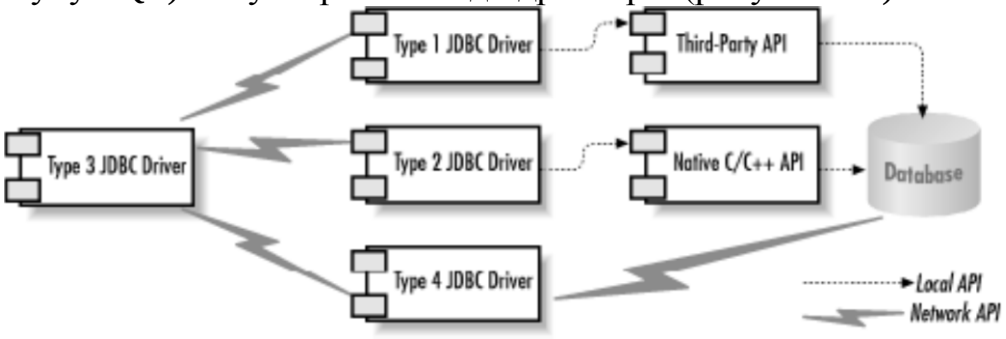

Рисунок 2.2 – Різні види JDBC драйверів

Тип 1 – драйвери цього типу використовують технологію bridge для доступу до бази даних. Міст JDBC-ODBC, що поставляється з JDK - хороший приклад цього типу драйверу. Он являє собою шлюз до ODBC API. Використання технології мосту потребує встановлення програмного забезпечення на клієнтських системах, що є неприйнятним для застосунків, які не дозволяють таких встановлень.

Тип 2 – це драйвери «рідного» API. Це значить, що драйвери містять java-код, який викликає методи на C чи C++, що надаються розробниками баз даних, що забезпечують доступ до баз даних. Це рішення також потребує встановлення програмного забезпечення на клієнтській системі.

Тип 3 – драйвери цього типу забезпечують клієнта загальним мережевим API, який транслює специфічний для бази даних доступ на серверний рівень. Драйвер JDBC клієнта використовує сокет щоб звернутися до проміжному програмному забезпеченню на сервері, яке транслює клієнтські запити в API необхідного драйвера. Це рішення потребує встановлення додаткового програмного забезпечення на клієнтській системі, і фактично з допомогою одного драйвера можна здійснювати робот з різними серверами баз даних.

Тип 4 – драйвери цього типу використовують мережевий протокол, вбудований в механізм бази даних. Ці драйвери звертаються безпосередньо до бази даних, використовуючи механізм java-сокетів. Це рішення реалізовано виключно на мові Java. Оскільки ці протоколи практично не документовані, драйвери цього типу практично завжди надаються розробниками баз даних.

В Java застосунку використовується інтерфейс JDBC, як фасад для безпосереднього доступу до бази даних, а реалізація JDBC виконується фактичний доступ до бази даних. Діаграма основних класів і інтерфейсів JDBC приведена на рисунку 2.3.

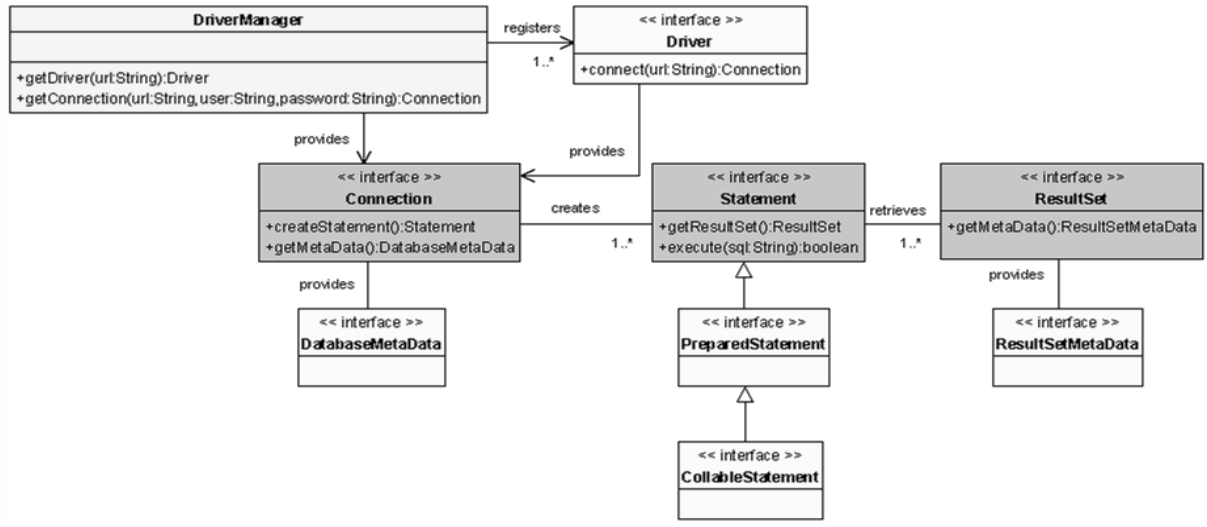

Рисунок 2.3 – Основні класи і інтерфейси JDBC API

JDBC Driver Manager – це основа JDBC-архітектури. Його первинні функції дуже прості - з’єднати java-програму і відповідний JDBC драйвер. Подальша робота проходить в три етапи:

– підключитися до БД;

- підготувати і передати SQL запит;
- отримання результатів і використання їх в застосунку.

2.2.2 Підключення до СУРБД

Джерело даних (data source) - це БД, яка зареєстрована с конкретним драйвером.

Для з'єднання з джерелом даних можна використовувати як міст JDBC-ODBC Bridge, що входить в поставку JDK, так і JDBC драйвер, що надається постачальниками БД.

З'єднання з джерелом даних виконується в три етапи:

- завантаження драйвера;
- підготовка URL джерела даних;
- підключення до джерела даних.

Спочатку необхідно завантажити драйвер, який повідомить класам JDBC, як взаємодіяти з джерелом даних.

У випадку використання JDBC драйвера конкретної БД, необхідна наявність відповідно бібліотеки класів, в випадку завантаження рідного драйвера для PostgreSQL, оператор буде виглядати наступним чином:

```
Class.forName(”org.postgresql.Driver”);
```

У випадку використання JDBC драйвера для вбудованої бази даних Derby запущеної в режимі сервера, необхідна наявність бібліотеки derbyclient.jar і завантаження драйвера:

```
Class.forName(”org.apache.derby.jdbc.ClientDriver”);
```

Методу *forName()*, в якості параметра передається повне ім'я класу драйвера. Цей метод може викидати виключення, тому його необхідно обгорнути в блок *try{}catch{}*.

```
try{ //завантаження драйвера
  Class.forName(”org.postgresql.Driver”);
}
catch(Exception e){ //обробка виключення
  e.printStackTrace ();
}
```

Операцію завантаження драйвера можна також здійснити з командного рядка. Але на практиці подібне використовується рідко. Завантаження драйвера в цьому випадку виконується з допомогою системної властивості jdbc.drivers, наприклад:

```
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver <ім'я програми>
```

Відповідно до встановлених правил Internet, БД і засоби її обслуговування ідентифікуються з допомогою URL (Uniform Resource Locator). В URL вказується, яким способом здійснюється підключення до джерела даних. Підключення може здійснюватися з допомогою мосту ODBC, с допомогою JDBC-драйвера БД і з допомогою мережевого сервісу. В загальному вигляді рядок URL має вигляд: jdbc:<постачальник>:<ім'я БД>, де під постачальником розуміється або ім'я конкретного драйвера чи мережевого сервісу, або деякого механізму встановлення з'єднання з БД.

У випадку використання JDBC-драйвера БД, URL джерела має вигляд: jdbc:<ім’я драйвера>:[//<хост>[:<порт>/]]<ім’я БД>.

Наприклад, при використанні драйвера для PostgreSQL, URL джерела має вигляд:

```
String url = ”jdbc:postgresql://localhost:5432/bibl”;
```

Наприклад, при використанні Derby в режимі сервера, URL джерела має вигляд:

```
'jdbc:derby:net://localhost:1527/bookstoredb:user=sa;password=sa;create=true';
```

Наприклад, при використанні Derby в режимі EMBEDDED, URL джерела має вигляд:

```
'jdbc:derby:embeded:bookstoredb:user=sa;password=sa;create=true';
```

Маючи підготований URL можна підключитися до джерела даних, використовуючи клас DriverManager. Директива запиту на підключення має вид:

```
String url = ”jdbc:postgresql://localhost:5432/bibl”;
String userName = ”customer”;
String password = ”Qwerty”;
Connectioncon=DriverManager.getConnection(url,userName,password);
```

DriverManager робить спробу з’єднання з драйвером. Для цього послідовно активізується кожний з завантажених драйверів, поки не буде знайдений той, який зможе обробити цей URL. В випадку якщо ні один з драйверів не зможе обробити даний URL буде створена виключна ситуація. Після роботи с БД з’єднання необхідно закрити, викликавши метод close() класу Connection. Далі приведений найпростіший приклад підключення до бази даних.

```
import java.sql.*;

public class SimpleConnection {
  static public void main(String args[]) {
    Connection connection = null;
    String url = "jdbc:postgresql://localhost:5432/bibl";
    String user = "customer", password = "Qwerty";
    try {
      Class.forName("org.postgresql.Driver");
    }
    catch( Exception e ) {
      e.printStackTrace( );
      return;
    }
    try {
      connection=DriverManager.getConnection(url,user,password);
      System.out.println("Connection successful!");
    }
    catch( SQLException e ) {
         e.printStackTrace( );
    }
```

```
    finally {
       if( connection != null ) {
          try { connection.close( ); }
          catch(SQLException e) {e.printStackTrace();}
       }
    }
  }
}
```

2.2.3 Отримання даних з бази даних і оновлення даних в базі даних

Коли встановлено з'єднання з джерелом даних, можна здійснювати оновлення і отримання даних з бази даних. Основна частина коду по роботі з БД пишуться з урахуванням того, що відомі інструкції, що надсилаються серверу БД, команди оновлення (INSERT, UPDATE, DELETE) чи команди, отримання даних (SELECT). Доступ до БД починається з створення об'єкта Connection. Цей об'єкт використовується для отримання іншого об'єкта, що реалізує інтерфейс java.sql.Statement, що пов'язаний з тою ж транзакцією, що і об'єкт Connection. Після використання одного або декількох об'єктів Statement можливо провести збереження (commit) або відкат (rollback) транзакції, пов'язаної з об'єктом Connection. Оскільки JDBC використовує різні методи для запитів і для оновлень, то дуже важливо знати якого типу SQL команда надсилається серверу БД. Основною відмінністю запиту на отримання даних від запиту на оновлення даних є те, що результатом запиту на отримання даних є об'єкт ResultSet, що представляє множину записів відповіді на запит, в той час, коли результатом запиту на оновлення даних є ціле число. Основними класами JDBC є Connection, Statement та ResultSet. Клас Statement самий основний з трьох класів, що представляють оператори SQL. Зазвичай проста транзакція БД використовує один з трьох методів класу Statement. Перший з трьох методів використовується для отримання результату запиту до БД на отримання даних ResultSet *executeQuery(String sql)* – приймає рядок запиту і повертає рядки результату. Другий метод використовується для оновлення даних в БД *int executeUpdate(String sql)* – приймає команду оновлення і повертає кількість успішно виконаних оновлень. Третій метод використовується у випадках, коли невідомий тип команди і результат, що повертається *boolean execute(String sql)* – приймає команду і повертає true, якщо результатом команди є множина рядків, в протилежному випадку повертає false. Клас ResultSet представляє один чи більше рядків, повернених запитом до БД. Цей клас просто надає набір методів для отримання даних з полів відповіді на запит до БД. Ці методи мають наступний формат: *type get type(int | String),* параметром може бути або номер колонки або її ім'я. Такий підхід можна використовувати для отримання даних різного типу з однієї і тієї ж колонки. Наприклад, якщо необхідно отримати значення поля, що має тип Date, у вигляді рядка, то необхідно використовувати метод getString(1) замість метода getDate(1). Оскільки ResultSet посилається тільки на один рядок результату в даний момент часу, то клас надає метод next(), що дозволяє перейти до наступного рядка відповіді на запит до БД. Якщо в наборі ще є рядки, то метод

next() повертає true, в протилежному випадку – false. SQL і Java мають серйозні відмінності в обробці null - значень. В Java ResultSet нема можливості представити SQL NULL значення для будь-якої числової колонки БД. Якщо БД дозволяє зберігати NULL значення для числових колонок, то необхідно використовувати метод wasNull() для перевірки на SQL NULL останній прочитаний результат, бо більшість драйверів поверне значення 0.

Всі три класи Connection, Statement і ResultSet мають метод close(). Не всі реалізації JDBC вимагають закривати об'єкти перед повторним використанням, але є реалізації, що потребують закривати об'єкти, тим самим звільняти ресурси сервера БД. Тому прийнято закривати об'єкти після закінчення роботи з ними. Важливо пам'ятати, що якщо закривати об'єкт Connection, то автоматично закриються всі пов'язані з ним об'єкти Statement. Аналогічно при закритті об'єктів Statement закриваються всі пов'язані з ним об'єкти ResultSet. При закритті з'єднання всі не збережені дані останньої відкритої транзакції будуть втрачені.

Розглянемо приклади виконання запитів з використанням JDBC. Наступний приклад демонструє виконання команди додання INSERT.

```
import java.sql.*;

public class Update {
  public static void main(String args[]) {
    Connection connection = null;
    if( args.length != 2 ) {
      System.out.println("Syntax: <java Update [number]  [string]>");
      return;
    }
    try {
      String driver = "org.postgresql.Driver";
      Class.forName(driver).newInstance();
      String url = "jdbc:postgresql://localhost:5432/testore";
      con = DriverManager.getConnection(url, "fooborg", "");
      Statement s = con.createStatement( );
      String test_id = args[0];
      String test_val = args[1];
      int update_count =
        s.executeUpdate("INSERT INTO test (test_id,test_val) "+
        "VALUES(" + test_id + ", '" + test_val + "')");
      System.out.println(update_count + " rows inserted.");
      s.close( );
    }
    catch( Exception e ) {
      e.printStackTrace( );
    }finally {
      if( con != null ) {
        try {
          con.close( );
        }catch( SQLException e ) {
          e.printStackTrace( );
```

```
        }
      }
    }
  }
}
```

За замовчуванням JDBC використовує властивість джерела даних – авто збереження (AUTO COMMIT). Тобто кожна зміна автоматично відображається в БД. Наступний приклад покаже, як можна відключити авто збереження, і управляти транзакцією, забезпечивши збереження результатів деякої SQL команд.

```
import java.sql.*;

public class UpdateLogic {
  public static void main(String args[]) {
    Connection connection = null;
    if( args.length != 2 ) {
      System.out.println("Syntax: <java UpdateLogic [number] [string]>");
      return;
    }
    try {
      String driver = "org.postgresql.Driver";
      Class.forName(driver).newInstance( );
      String url = "jdbc:postgresql://localhost:5432/testore";
      Statement s;
      con = DriverManager.getConnection(url, "fooborg", "");
      con.setAutoCommit(false);
      s = con.createStatement( );//create the first statement
      s.executeUpdate("INSERT INTO test (test_id,test_val) "+
        "VALUES(" + args[0] + ", '" + args[1] + "')");
      s.close( ); // close the first statement
      s=con.createStatement();// create the second statement
      s.executeUpdate("INSERT into test_desc (test_id, test_desc) " +
        "VALUES(" + args[0] + ", '" + args[1] + "')");
      con.commit( ); // commit the two statements
      System.out.println("Insert succeeded.");
      s.close( ); // close the second statement
    } catch( SQLException e ) {
      if( con != null ) {
        try {
          con.rollback();
        } catch( SQLException e ) {
        }
      }
      e.printStackTrace( );
    } finally {
      if( con != null ) {
        try {
          con.close( );
        } catch( SQLException e ) {
          e.printStackTrace( );
```

```
        }
      }
    }
  }
}
```

Клас ResultSet представляє відповідь на запит до БД. існує декілька типів об'єктів цього класу. Вказівка типу результату запиту відбувається при його створення, тобто при викликанні методу createStatement() класу Connection. Цей метод має декілька реалізацій *Statement createStatement(int resultSetType, int resultSetConcurrency).* При викликанні без параметрів він транслюється у виклик: *createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY).*

Для переміщення по рядкам результату запиту існує метод next(). Спочатку курсор займає вихідну позицію «над» першим рядком, тобто необхідно викликати метод next() перед тим як використовувати об'єкт ResultSet. Окрім next() JDBC надає наступні методи переміщення по рядкам результату запиту: previous(), beforeFirst( ), first( ), beforeLast(), last( ), absolute( ), і relative( ).

Для визначення місця положення курсору існують наступні методи isFirst(), isLast(), isBeforeFirst(), isAfterLast(), і getRow( ). Всі методи крім getRow() повертають значення типа boolean, а getRow() – повертає номер поточного рядка.

Нижче приведений приклад, що демонструє переміщення по результату запиту.

```
import java.sql.*;

public class ReverseSelect {
  public static void main(String argv[]) {
    Connection con = null;
    try {
      String url = "jdbc:postgresql://localhost:5432/testore";
      String driver = "org.postgresql.Driver";
      Statement stmt;
      ResultSet rs;
      Class.forName(driver).newInstance( );
      con = DriverManager.getConnection(url, "fooborg", "");
      stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
      rs = stmt.executeQuery("SELECT * from test ORDER BY test_id");
      System.out.println("Отриманий результат:");
      rs.afterLast( );
      while(rs.previous( )) {
        int a;
        String str;
        a = rs.getInt("test_id");
        if( rs.wasNull( ) ) {
          a = -1;
        }
```

```
        str = rs.getString("test_val");
        if( rs.wasNull( ) ) {
          str = null;
        }
        System.out.print("\ttest_id= " + a);
        System.out.println("/str= '" + str + "'");
      }
      System.out.println("Все.");
    } catch( Exception e ) {
      e.printStackTrace( );
    } finally {
      if( con != null ) {
        try {
          con.close( );
        } catch( SQLException e ) {
          e.printStackTrace( );
        }
      }
    }
  }
}
```

2.2.4 Збирання проекту з використанням Gradle

Системи автоматизованого збирання проектів використовуються для спрощення і автоматизації процесів життєвого циклу застосунків. Вони допомагають автоматизувати процес збирання, запуску і тестування застосунків, а також допомагають зробити більш зручним управління залежностями проекту. Починаючи з системи автоматизованого збирання Maven був створений репозитарій Maven Central, що містить велику кількість вихідних кодів різноманітних бібліотек, які можна додавати в залежності проекту. Дані бібліотеки автоматично відвантажуються в проект при збиранні застосунку. В такому випадку нема необхідності шукати необхідні бібліотеки по різноманітним сайтам розробників. Найсучаснішою системою автоматизованого збирання є Gradle. Для того, щоб мати можливість збирати проекти з використанням Gradle необхідно завантажити архів зі сторінки https://gradle.org/install/, розпакувати його і прописати кореневий каталог gradle в змінній середовища PATH. Це необхідно зробити, щоб мати можливість виконувати команду gradle з терміналу без вказування кожен раз кореневого каталогу gradle. Перевагою gradle перед іншими системами автоматизованого збирання є можливість підключення до скрипта великої кількості плагінів, до додають до скрипта додаткові цілі, які в Gradle називаються завданнями (task).

Приклад підключення плагіна приведений нижче:

```
apply plugin: 'java'
```

Для вказування залежностей від сторонніх бібліотек слугує елемент dependencies, а для додання зовнішніх репозитаріїв слугує елемент repositories Приклад залежності від сторонньої бібліотеки приведений нижче:

```
repositories {
```

```
  mavenCentral()
}
dependencies {
  runtime group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
}
```

Окрім завдань, які додаються з використанням плагінів можна створювати свої власні завдання. Приклад власного завдання приведений нижче:

```
task hello {
  doLast {
    println "hello"
  }
}
```

Для того, що виконати дане завдання треба передати ім’я цього завдання на вхід команди gradle:

gradle hello

Більш детально ознайомитися з системою автоматизованого збирання Gradle можна на сайті http://docs.gradle.org.

2.2.5 Робота з вбудованими базами даних

Бази даних в пам’яті покладаються на системну пам’ять, а не на дисковий простір для збереження даних. Оскільки доступ до пам’яті швидший, ніж до диску, ці бази даних швидші. Звичайно, ми можемо використовувати базу даних в пам’яті тільки в тих застосунках і сценаріях, де нема необхідності зберігати дані чи виконувати тести швидше. Використання вбудованих баз даних не потребує налаштування зовнішньої бази даних, а дані створюються при запуску процесу і видаляються при його завершенні.

Існує декілька вбудованих в пам'ять баз даних, які можна використовувати в застосунку такі як H2, HSQLDB, Apache Derby, SQLite.

Зазвичай, для того, щоб почати працювати з базою даних вбудованою в пам'ять достатньо завантажити бібліотеку у вигляді jar-архіву і знати рядок підключення до бази даних. Зазвичай бібліотеку можна завантажити з репозитарія Maven Central. Нижче приведені конфігурації gradle і рядки підключення найбільш відомих вбудованих баз даних:

```
//H2
dependencies {
  runtime group: 'com.h2database', name: 'h2', version: '1.4.194'
}
//Налаштування для H2 DB
driverClassName=org.h2.Driver
url=jdbc:h2:mem:myDb;DB__CLOSE__DELAY=-1
username=sa
password=sa

//HSQLDB
dependencies {
  runtime group: 'org.hsqldb', name: 'hsqldb', version: '2.3.4'
```

```
}
//Налаштування для HSQLDB DB
driverClassName=org.hsqldb.jdbc.JDBCDriver
url=jdbc:hsqldb:mem:myDb
username=sa
password=sa

//Apache Derby
dependencies {
  runtime group: 'org.apache.derby', name: 'derby', version: '10.13.1.1'
}
//Налаштування для Apache Derby DB
driverClassName=org.apache.derby.jdbc.EmbeddedDriver
url=jdbc:derby:memory:myDb;create=true
username=sa
password=sa

//SQLite
dependencies {
  runtime group: 'org.xerial', name: 'sqlite-jdbc', version: '3.16.1'
}
//Налаштування для SQLite DB
driverClassName=org.sqlite.JDBC
url=jdbc:sqlite:memory:myDb
username=sa
password=sa
```

Після завантаження драйверу з репозитарія алгоритм роботи з вбудованою базою даних схожий на алгоритм роботи з звичайною базою даних. Приклад створення підключення для Apache Derby DB приведений нижче:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection c = DriverManager.getConnection("jdbc:derby:testdb1;create=true");
```

Оскільки при створенні підключення в базі даних відсутні таблиці і дані в них, то їх необхідно туди додати. Для автоматизації цього процесу можна використати клас бібліотеки Spring – EmbeddedDatabaseBuilder. Даний клас допомагає виконати SQL команди, що знаходяться в файлах ресурсів. На даний момент підтримуються бази даних H2, HSQLDB та Apache Derby Приклад використання класу EmbeddedDatabaseBuilder для створення таблиць і додання в них записів приведений нижче:

```
EmbeddedDatabase embeddedDatabase = new EmbeddedDatabaseBuilder()
  .setType(EmbeddedDatabaseType.H2)
  .addScript("ua/cn/stu/resources/schema.sql")
  .addScript("ua/cn/stu/resources/init.sql").build();
```

Файл schema.sql містить запити на створення таблиць бази даних, а файл init.sql містить запити на додання записів в ці таблиці.

2.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення застосунку для роботи за базою даних з допомогою якого можна

переглядати записи в базі даних і додавати нові записи. В якості прикладу був обраний об’єкт предметної області «інтернет-магазин» – продукт. Додатково до аспектів взаємодії з базою даних в даній лабораторній роботі розглядаються аспекти роботи з системою автоматизованого збирання проектів – Gradle. Система автоматизованого збирання проектів Gradle використана в даній лабораторній роботі для завантаження з репозитарія Maven необхідних бібліотек. Для використання системи автоматизованого збирання проектів її необхідно попередньо встановити або розпакувати в певне місце на жорсткому диску і додатково додати кореневий каталог gradle до змінної середовища PATH, що дасть змогу виконувати команду gradle з консолі без необхідності вказування кореневого каталогу gradle. Для створення графічного інтерфейсу в даній лабораторній роботі використана бібліотека Swing, а для спрощення створення графічного інтерфейсу використовується плагін WindowBuilder. В якості системи управління базою даних в лабораторній роботі використана СУБД PostgeSQL 8.3.

1. Завантажте gradle архів з сайту https://gradle.org/install/ та розпакуйте його в будь яке місце на жорсткому диску, додайте кореневий каталог gradle до змінної оточення PATH.
2. Відкрийте Eclips та встановіть плагіни WindowBuilder та BuildShip Gradle Integration. WindowBuilder потрібний для створення графічного інтерфейсу на базі бібліотек Swing. BuildShip Gradle Integration для підсвічування синтаксису gradle.
3. Встановіть СУБД PostgreSQL. Вона використовується в лабораторній роботі в якості прикладу, але замість неї можна використовувати іншу мережеву СУБД, наприклад, MySQL, MS SQL Server або Oracle СУБД. В залежності від обраної СУБД необхідно буде змінити рядок підключення і завантажити необхідний JDBC драйвер.
4. Створіть новий Java проект та в каталозі src створіть пакет ua.cn.stu.main, створіть каталог lib.
5. Після встановлення плагіну WindowBuilder в Eclipse буде додана можливість створення Application Window. Створіть Application Window з іменем DatabaseExplorer, як показано на рисунку 2.4.

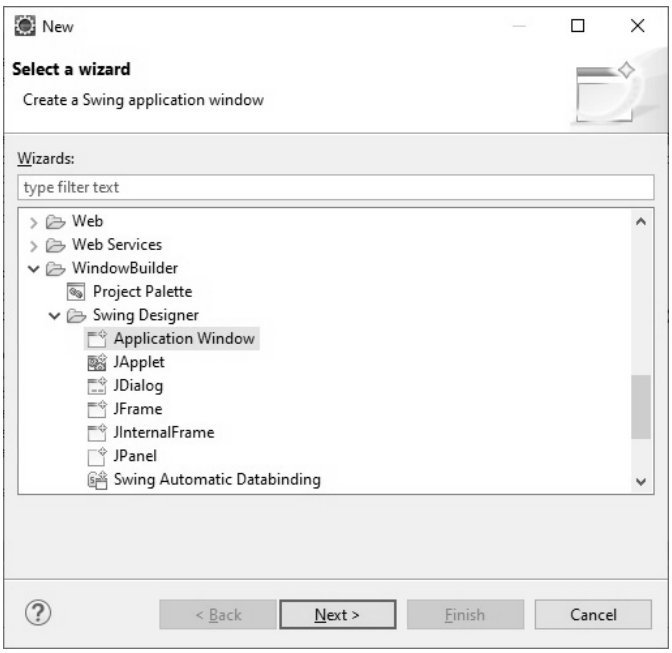

Рисунок 2.4 – Вікно створення Application Window

6. Створіть в СУБД PostgreSQL нову базу даних shop, а в базі даних shop створіть табличку з використанням запиту представленого нижче:

```
CREATE TABLE PRODUCT (
  PRODUCT_ID SERIAL NOT NULL,
  PRODUCT_NAME VARCHAR(20) NOT NULL,
  PRODUCT_DESCRIPTION VARCHAR(20) NOT NULL,
  PRIMARY KEY (PRODUCT_ID)
);
```

7. В корені проекту створіть файл build.gradle з контентом:

```
apply plugin: 'java'
//Репозитарій з якого буде завантажено бібліотеки
repositories {
  mavenCentral()
}

sourceSets {
  main {
    java {
      srcDir 'src'
    }
  }
}

//Список бібліотек, які необхідно завантажити
dependencies {
  runtime group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
}

//Клас, що буде головним при запуску програми
jar {
  manifest {
    attributes 'Main-Class': 'ua.cn.stu.main.DatabaseExplorer'
```

```
  }
}

//Команда для копіювання класів в каталог lib
task copyToLib(type: Copy) {
  from configurations.runtime
  into "lib"
}

//Вказування залежності завдання build від завдання copyToLib
project.afterEvaluate {
  tasks.build.dependsOn(copyToLib)
}
```

8. Відкрийте термінал Windows, перейдіть в корінь проекту та запустіть команду gradle build, що спричинить компіляцію класів проекту, завантаження бібліотек з репозитарія Maven і копіювання бібліотек в каталог lib. Поновіть каталог lib. В ньому повинна з'явитися бібліотека postgresql-8.3-603.jdbc4.jar. Додайте дану бібліотеку до Java Build Path (Build Path -> Configure Build Path -> Libraries -> Add JARs). В залежності від встановленої версії СУБД необхідно змінити версію драйвера в файлі build.gradle в розділі dependencies. В даному випадку в dependencies вказана версія JDBC драйвера 8.3-603.jdbc4.

9. Створіть пакет ua.cn.stu.domain. Додайте в пакет клас Product з кодом як показано нижче:

```
public class Product {

  private Long id;
  private String name;
  private String description;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public Long getId() {
    return id;
  }
```

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

  @Override
  public String toString() {
    return "id - " + id + ", name - " + name + ", description - " +
      description;
  }
}
```

10. Створіть графічний інтерфейс застосунку з використанням плагіна WindowBuilder як показано на рисунку 2.5.

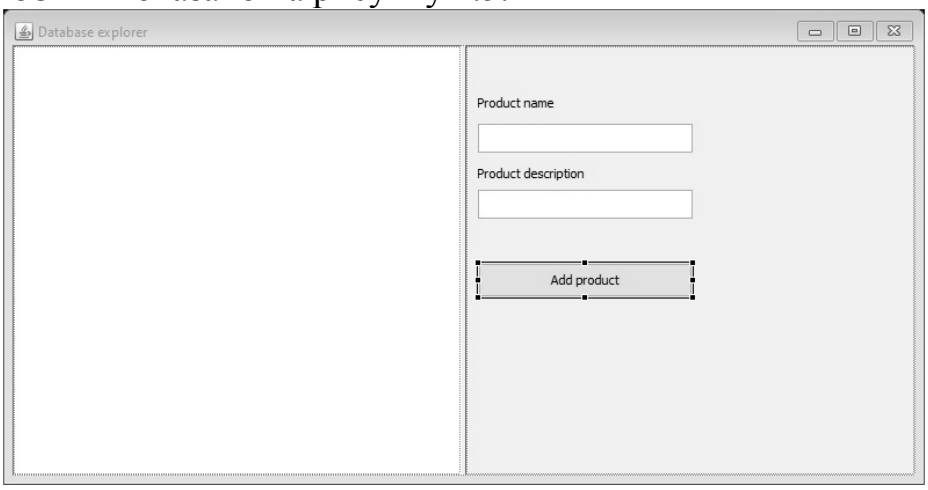

Рисунок 2.5 – Графічний інтерфейс застосунку

Додайте два графічних компоненти Text Field, один графічний компонент Button для вводу даних і графічний компонент List для виводу даних. Замість вказаних елементів можна використовувати будь-які інші елементи для вводу і виводу даних.

11. Додайте в клас DatabaseExplorer метод для підключення до бази даних:

```
private static Connection connectToDatabase(String url, String name, String
password) throws SQLException, ClassNotFoundException {
  Class.forName("org.postgresql.Driver");
  Connection connection = DriverManager.getConnection(url, name, password);
  return connection;
}
```

12. Додайте в клас DatabaseExplorer метод для отримання всіх записів з таблички product:

```
private static List<Product> getAllProducts(Connection connection) throws
SQLException {
  List<Product> productList = new ArrayList<Product>();
  PreparedStatement preparedStatement =
  connection.prepareStatement("select * from product");
  ResultSet result = preparedStatement.executeQuery();
  while (result.next()) {
    Product product = new Product();
    product.setId(result.getLong(1));
```

```
    product.setName(result.getString(2));
    product.setDescription(result.getString(3));
    productList.add(product);
  }
  return productList;
}
```

13. Додайте в клас DatabaseExplorer метод для додання в табличку product нового запису:

```
private static void addProduct(Connection connection, String productName,
String productDescription) throws SQLException {
  PreparedStatement preparedStatement = connection.prepareStatement(
    "insert into product(product_name, product_description) values (?, ?)");
  preparedStatement.setString(1, productName);
  preparedStatement.setString(2, productDescription);
  preparedStatement.execute();
}
```

14. Додайте в клас DatabaseExplorer поля, що будуть містити об’єкт підключення до бази даних і об’єкт моделі списку, в який буде виводитись вміст таблички:

```
private static Connection connection = null;
private static DefaultListModel<Product> produListModel;
```

15. Додайте в головний метод програми (клас DatabaseExplorer) код, що буде виконувати підключення до бази даних і виводити вміст таблички бази даних в візуальний компонент List. Після внесення змін код голосного методу буде виглядати наступним чином:

```
public static void main(String[] args) {
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      try {
        DatabaseExplorer window = new DatabaseExplorer();
        window.frmDatabaseEplorer.setVisible(true);
        connection = connectToDatabase(
          "jdbc:postgresql://localhost:5432/shop", "postgres", "111111");
        List<Product> productList = getAllProducts(connection);
        produListModel.removeAllElements();
        for (Product product : productList) {
          produListModel.addElement(product);
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  });
}
```

16. Додайте в обробник натиснення кнопки (клас DatabaseExplorer) код для додання нового запису в табличку бази даних і виведення результату в список. Код обробника кнопки буде виглядати наступним чином після внесення змін:

```
JButton btnNewButton = new JButton("Add product");
btnNewButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    addProduct(connection, textField1.getText(), textField2.getText());
    List<Product> productList = getAllProducts(connection);
    produListModel.removeAllElements();
    for (Product product : productList) {
      produListModel.addElement(product);
    }
  }
});
```

17. Виконайте додаткову ініціалізацію графічного компоненту List в методі initialize класу DatabaseExplorer , та передайте в нього в якості параметру тип Product:

```
JList<Product> list = new JList<Product>();
produListModel = new DefaultListModel<Product>();
list.setModel(produListModel);
```

18. Перевірте працездатність програми.

19. Удосконалимо програму створену на попередніх етапах з використанням Spring JDBC template. Використання Spring JDBC template дозволяє уникнути дублювання коду і при виконанні запитів одразу отримувати колекції об’єктів замість екземпляру класу ResultSet. Для використання Spring JDBC template нам необхідні додаткові бібліотеки, які треба прописати в build.gradle файлі. Змініть елемент dependencies даного файлу як показано нижче:

```
dependencies {
  runtime group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
  runtime group: 'org.springframework', name: 'spring-jdbc',
  version: '5.2.4.RELEASE'
}
```

20. Відкрийте термінал Windows, перейдіть в корінь проекту та запустіть команду gradle build, що спричинить компіляцію класів проекту, завантаження бібліотек з репозитарія Maven і копіювання бібліотек в каталог lib. Поновіть каталог lib. Додайте всі бібліотеки, що з’явилися в Java Build Path.

21. Для перетворення запису ResultSet об’єкт додамо клас, що буде виконувати перетворення. Додамо цей клас в пакет:

```
public class ProductMapper implements RowMapper<Product> {

  @Override
  public Product mapRow(ResultSet resultSet, int i) throws SQLException {
    Product product = new Product();
    product.setId(resultSet.getLong("product_id"));
    product.setName(resultSet.getString("product_name"));
    product.setDescription(resultSet.getString("product_description"));
    return product;
  }
}
```

22. Додайте додаткове поле, що буде містити об’єкт класу DriverManagerDataSource:

```
private static DriverManagerDataSource dataSource;
```

23. Додайте в клас DatabaseExplorer метод для створення об’єкту DriverManagerDataSource. Цей об’єкт є аналогом класу Connection JDBC і містить набір методі, що дозволяють взаємодіяти з базою даних:

```
private static DriverManagerDataSource connectToDatabaseJDBCTemplate(String
url, String name, String password) throws ClassNotFoundException {
  Class.forName("org.postgresql.Driver");
  DriverManagerDataSource dataSource = new DriverManagerDataSource();
  dataSource.setDriverClassName("org.postgresql.Driver");
  dataSource.setUrl(url);
  dataSource.setUsername(name);
  dataSource.setPassword(password);
  return dataSource;
}
```

24. Додайте в клас DatabaseExplorer метод для додання нового запису в табличку product з допомогою Spring JDBC template:

```
private static void addProductJDBCTemplate(DriverManagerDataSource dataSource,
String productName, String productDescription) {
  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
  jdbcTemplate.update("insert into product(product_name,
    product_description) values (?, ?)", productName, productDescription);
}
```

25. Додайте в клас DatabaseExplorer метод для отримання всіх записів таблички product з допомогою Spring JDBC template:

```
private static List<Product> getAllProductsJDBCTemplate(DriverManagerDataSource
dataSource) {
  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
  return jdbcTemplate.query("select * from product", new ProductMapper());
}
```

26. Замініть виклик методів connectToDatabase та getAllProducts на методи connectToDatabaseJDBCTemplate і getAllProductsJDBCTemplate в головному методі класу DatabaseExplorer.

27. Замініть виклик методів addProduct і getAllProducts на addProductJDBCTemplate і getAllProductsJDBCTemplate в обробнику натиснення на кнопку класу DatabaseExplorer.

28. Протестуйте працездатність застосунку.

29. Вдоскональте застосунок відповідно до завдання на лабораторну роботу.

2.4 Завдання на лабораторну роботу

Вдосконалити застосунок, описаний в ході виконання лабораторної роботи, щоб він міг працювати з двома табличками БД, на Відміну від застосунку, описаного в лабораторній роботі. Предметну область необхідно вибрати з таблиці 2.1. Номер варіанта визначається по останній цифрі залікової книжки. Таблички повинні бути зв’язані відношенням один до багатьох.

Таблиця 2.1. Варіанти завдань

Номер варіанту	Предметна область
1	Піцерія (Офіціант, Відвідувач)
2	Крамниця (Постачальник, Товар)
3	Супермаркет (Продавець, Товар)
4	Відділення поліції (Поліцейський, Порушник)
5	Лікарня (Лікар, Пацієнт)
6	Кінотеатр (Кінозал, Відвідувач)
7	Корабель (Каюта, Пасажир)
8	Підприємство (Відділ, Співробітник)
9	Бібліотека (Книга, Читач)

2.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

2.6 Контрольні питання

1. Розказати про архітектуру JDBC;
2. Основні класи і інтерфейси JDBC;
3. Назвіть основні функції Driver Manager;
4. Назвіть порядок підключення до джерела даних;
5. Назвіть порядок встановлення з’єднання з джерелом даних;
6. Отримання даних різних типів з об’єкта ResultSet;
7. Що повертає ResultSet;
8. Назвіть основні методи для переміщення по рядкам результату запиту;
9. Чим Statement відрізняється від PreparedStatement;
10. Чому краще використовувати PreparedStatement замість Statement.

2.7 Завдання до захисту лабораторної роботи

1. Зберегти табличку бази даних в XML файл.
2. Завантажити дані з XML файлу в табличку бази даних з лабораторної роботи.
3. Зберегти табличку бази даних в JSON файл.
4. Завантажити дані з JSON файлу в табличку бази.
5. Зберегти табличку бази даних в CSV файл.
6. Завантажити дані з CSV файлу в табличку бази.
7. Відобразити структуру бази даних в текстовому вигляді.

8. Відобразити структуру бази даних в графічному вигляді з використанням графічних бібліотек.

9. Замінити базу даних, використану в лабораторній роботі, на вбудовану базу даних, наприклад Derby або H2.

10. Замінити базу даних використану в лабораторній роботі на NoSQL базу даних.

3 ЛАБОРАТОРНА РОБОТА №3
ВИВЧЕННЯ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ

3.1 Мета роботи

Вивчити технологію об'єктно-реляційного відображення Hibernate, отримати практичні навички по створенню застосунків, що використовують об'єктно-реляційне відображення для роботи за базою даних.

3.2 Теоретичні відомості

3.2.1 Об'єктно-реляційне відображення

При розробці застосунку, який вимагає здійснення доступу до даних, необхідно спростити розробку такого застосунку, збільшивши ефективність і швидкість роботи з отриманими даними. На рисунку 3.1 приведений один з стандартних варіантів організації доступу до даних з використанням драйвера JDBC.

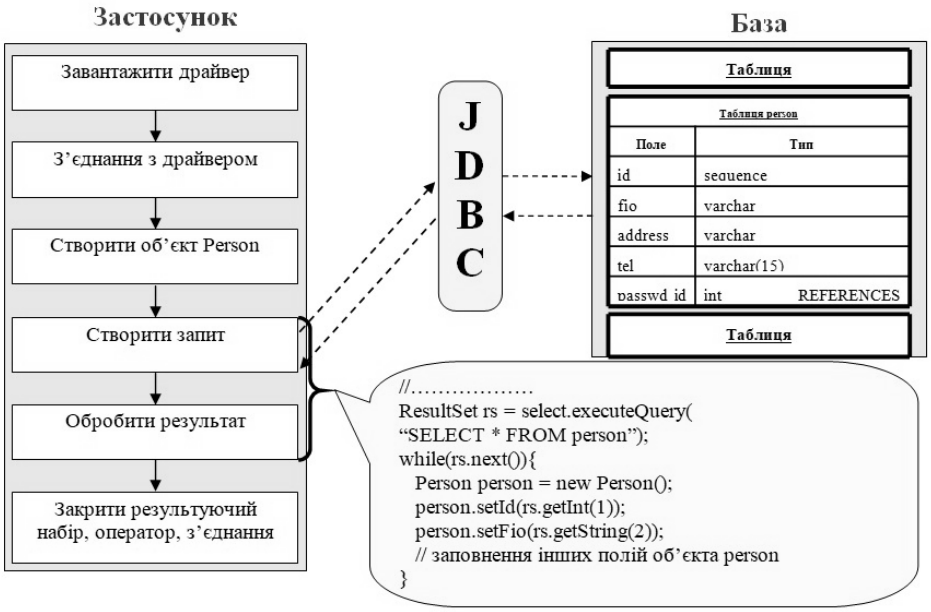

Рисунок 3.1 – Доступ до БД з Java застосунку

Як видно з рисунку 3.1, необхідно пройти декілька етапів, для того, щоб отримати дані. Необхідно вручну формувати об'єкт з отриманих даних. Об'єктно-реляційне відображення (ORM) - це техніка програмування, яка зв'язує реляційну базу даних з концепціями об'єктно-орієнтованого програмування і створює віртуальну базу даних об'єктів. Ця техніка програмування з'явилася з наступних причин:

1. Робота переноситься з рівня роботи з записами бази даних на інший рівень – рівень роботи з об'єктами.

2. Велика гнучкість організації доступу до даних (об'єкт може складатися з полів різних таблиць).

3. Підтримка доступу декількох користувачів до одних і тих же об'єктів (робота з persistent об'єктами).

4. Однотипний доступ до різних баз даних – розробник не приймає до уваги сервер баз даних, з яким він працює (PostgreSQL, MySQL, Oracle і т д.).

5. Спрощення організації зв'язку між об'єктами – посилання на об'єкти.

6. Зменшення часу доступу до даних за рахунок використання механізму кешування.

На даний момент існують різні фреймворки, що використовують дану техніку програмування, наприклад Hibernate, Java Data Objects (JDO), TopLink та Open JPA. JPA не є новою технологією. Скоріше, це збірка ідей найкращих з наявних технологій, таких як Hibernate, TopLink і JDO. Як результат JPA є стандартизованою специфікацією, що входить в JDK, що дозволяє будувати шар збереження даних незалежно від будь-яких конкретних провайдерів. Тобто реалізацій специфікації JPA може бути багато, однією з таких, наприклад є фреймворк Open JPA чи той же Hibernate.

3.2.2 Архітектура Hibernate

Архітектура Hibernate представлена на рисунку 3.1.

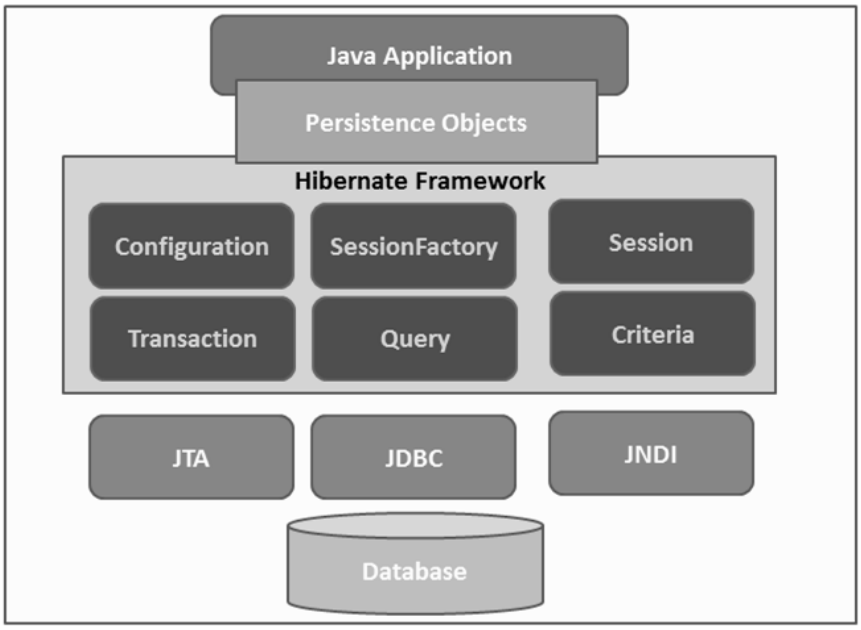

Рисунок 3.2 – Архітектура Hibernate

Давайте розглянемо окремо кожний з елементів Hibernate, які вказані на архітектурі.

Об'єкт *Transaction* є робочою одиницею взаємодії з базою даних. В Hibernate транзакції обробляються менеджером транзакцій.

SessionFactory – самий важливий і самий важкий об'єкт (зазвичай створюється в єдиному екземплярі при завантаженні застосунку). Необхідно створити як мінімум одну фабрику *SessionFactory* для кожної БД, кожна з яких конфігурується окремими конфігураційним файлом.

Об'єкт *Session* – використовується для отримання фізичного з'єднання з БД. Зазвичай сесія створюється за необхідністю, а після виконання необхідних дій закривається. Цей об'єкт використовується для отримання, зміни і видалення об'єктів.

Об’єкт *Query* використовує мову запитів HQL або SQL для отримання або запису даних з або в базу даних. Екземпляр запиту використовується для зв’язування параметрів запиту, обмеження кількості результатів, які будуть повернуті і для виконання запитів.

Об’єкт *Configuration* використовується для створення об’єкту SessionFactory з використанням конфігураційного XML файлу, який пояснює як обробляти об’єкт *Session*.

Об’єкт *Criteria* використовується для виконання об’єктно-орієнтованих запитів для отримання об’єктів.

Послідовність створення об’єктів Hibernate представлена нижче:

```
//Створення об’єкту конфігурації
Configuration configuration = new Configuration().configure();
StandardServiceRegistryBuilder serviceRegistryBuilder = new
StandardServiceRegistryBuilder();
serviceRegistryBuilder.applySettings(configuration.getProperties());
ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
//Створення фабрики сесій
SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistry);
//Відкриття сесії
Session session = sessionFactory.openSession();
//Створення об'єкту сутності
Group group  = new Group();
group.setName("Group name");
//Відкриття транзакції
Transaction transaction = session.beginTransaction();
//Збереження сутності в базу даних
session.saveOrUpdate(group);
//Закриття транзакції
transaction.commit();
//Запит на отримання даних
SQLQuery query = session.createSQLQuery("select * from
grouptable").addEntity(GroupEntity.class);
List<GroupEntity> groupList = query.list();
```

3.2.3 Сутності JPA (збережені об’єкти)

Сутністю в JPA може бути будь-який Java клас, який, однак, повинен відповідати певним умовам:

1. Наявність конструктора за замовчуванням (конструктора без параметрів). Специфікація JPA вимагає, щоб класи всіх збережених об’єктів мали конструктор замовчуванням в Java, тобто конструктор без параметрів. Цей конструктор може бути public чи protected. Так як компілятор Java автоматично створює конструктор за замовчуванням, коли не визначено інших конструкторів, то це обмеження діє тільки на класи, які мають додаткові конструктори.

2. Класи не final. Класи збережених об’єктів не можуть бути final. Також не можуть бути final методи цих класів.

3. Наявність полів ідентифікації. Все класи збережених об’єктів повинні містити один одне або декілька полів, які разом формують унікальний ключ для сутності. Ці поля називаються identity або primary key.

4. Спадкування. Класи збережених об’єктів не можуть спадкуватися від деяких класів, які залежать від операційної системи, такі як java.net.Socket і java.lang.Thread. Якщо клас збереженого об’єкта спадкується від класу не збереженого об’єкта, тоді поля не збереженого суперкласу не можуть бути збереженими. Всі класи в дереві спадкування повинні мати один і той же тип ідентифікації.

3.2.4 Метадані сутностей

Для вказування метаданих сутностей існує два способи: анотації Java та XML файли. При використанні XML файлів необхідно створювати файл orm.xml і помістити його в каталог META-INF разом з файлом persistence.xml. В цьому файлі, використовуючи спеціальні теги, необхідно вказати параметри відображення для сутностей і їх полів. Другим способом опису метаданих є використання анотацій Java. В цьому випадку спеціальні анотації вказуються над класами і їх полями, визначаючи відображення сутностей в базу даних.

JPA підтримує використання або одного з цих способів, або обох способів одночасно. При використанні обох способів одночасно опис в файлі XML має більш високий пріоритет, ніж анотації. В даній лабораторній роботі ми будемо використовувати анотації. Далі розглянемо опис основних анотацій, що використовуються для опису метаданих відображення. Всі анотації можна розділити на анотації сутностей і анотації полів. Розглянемо більш детально анотації сутностей.

Анотація *@Entity* – вказує, що даний клас є збереженою сутністю. Необхідно вказувати дану анотацію над кожною сутністю. Опціонально можна вказати одну властивість name, яка буде використовуватись для звернення до сутності в запитах. За замовчуванням іменем сутності є коротке ім’я класу.

```
@Entity
public class Group {
…
}

@Entity(name="Student")
public class Student {
…
}
```

Анотація *@Table* – не обов’язкова анотація. Вказує ім’я таблиці, в яку будуть зберігатися сутності. Також можна вказати імя схеми в базі даних. За замовчуванням використовується таблиця, яка відповідає короткому імені класу.

```
@Table(name="TABLE_GROUP", schema="DEFAULT_SCHEMA")
public class Group {
…
}
```

За замовчуванням в JPA всі поля, які є примітивними типами (int, float, …), обгортками примітивних типів (Integer, Float, …), класами String, byte[], Byte[], char[], Character[], BigDecimal, BigInteger, Date, Calendar, Date, Timestamp, чи будь-якими іншими об’єктами , що реалізують інтерфейс Serializable, будуть зберігатися в базу даних в колонки, що відповідають іменам полів, якщо не визначено інше. Поля, визначені як static, transient, чи final зберігатися в базу даних не будуть. Додатково для кожного поля можна вказати колонку, спосіб отримання даних, чи є поле ідентифікатором чи його взагалі не треба зберігати в базу даних. Також з допомогою анотацій можна вказати зв’язки об’єкта з другими об’єктами. Розглянемо більш детально анотації полів.

Анотація *@Transient* – вказує, що дане поле не буде зберігатися в базу даних.

```
@Transient
private int someInfo;
```

Анотація *@Id* – вказує, що дане поле є простим ідентифікатором (primary key) сутності.

```
@Id
private int id;
```

Анотація *@Generated* – вказує, що поле ідентифікатора буде автоматично генеруватися, використовуючи один із способів. Спосіб можна вказати в полі strategy. Допустимі значення:

- AUTO – автоматичний вибір стратегії ключів (за замовчуванням);
- IDENTITY – використання стратегії бази даних;
- SEQUENCE – використання джерела даних для алгоритму послідовності;
- TABLE – використання окремої таблички для алгоритму послідовності.

```
@Id
@GeneratedValue
private int id;
```

Анотація *@Basic* – вказує, що дане поле буде зберігатися в базу даних. Зазвичай дану анотацію не вказують, крім тих випадків, коли необхідно вказати додаткові параметри відображення. Fetch дозволяє вказати стратегію отримання даних (EAGER – дані вибираються одразу в запиті (за замовчуванням), LAZY – дані вибираються при першому зверненні до них). Властивість optional дозволяє вказати, чи дозволені null значення в колонці таблички і в полі відповідно. За замовчуванням optional=true, що значить, що null значення дозволені.

```
@Basic
private String fio;

@Basic(fetch=LAZY, optional = false)
private String name;
```

Анотація *@Temporal* – для позначення, що поле є типом дати і часу. Допустимі параметри анотації: DATE (тільки дата) TIME (тільки час) TIMESTAMP (дата і час).

```
@Temporal(TIMESTAMP)
private Calendar startDate;

@Temporal(DATE)
private Calendar birthDate;
```

Анотація *@ManyToOne* – коли об’єкт А містить посилання на об’єкт В і об’єкт АА також може посилатися на В, тоді маємо відношення багато-до-одного. Для вказування такого зв’язку зі сторони «багато» використовується дана анотація. Властивості fetch і optional каті ж, як і для попередньої анотації. Властивість, що вказує поведінку при каскадному доступі до об’єкта, cascade може приймати наступні значення:

- PERSIST – зберігає зв’язаний об’єкт при збереженні поточного об’єкта;
- MERGE – приєднувати зв’язаний об’єкт при приєднанні поточного до менеджера;
- REMOVE – видаляти зв’язаний об’єкт при видаленні поточного об’єкта;
- REFRESH – оновлювати стан зв’язаного об’єкта при оновленні стану поточного об’єкта;
- ALL – виконувати всі каскадні зміни зі зв’язаним об’єктом.

```
@ManyToOne(cascade={ PERSIST, MERGE, REMOVE, REFRESH })
private Group group;

@ManyToOne(cascade=ALL)
private Group group;

@ManyToOne(fetch = LAZY)
private Group group;

@ManyToOne(
        optional = true)
private Group group;
```

Анотація *@OneToMany* – якщо об’єкт А посилається на декілька об’єктів В, а кожний об’єкт В посилається тільки на один об’єкт А, тоді маємо зв'язок один-до-багатьох. Для вказування такого зв’язку зі сторони «один» використовується дана анотація. Якщо з іншої сторони також присутній зв'язок (тобто у об’єкта В є посилання на А і визначена анотація @ManyToOne), тоді необхідно вказати, що даний зв'язок двонаправлений. Для цього використовується властивість mappedBy, в якій вказується ім’я властивості на іншій стороні (тобто в об’єкті В), яке посилається на даний клас. Інші властивості, можливі в даній анотації, cascade і fetch, такі ж, як і для попередніх анотацій.

```
@OneToMany
```

```
private Set<Student> students;

@OneToMany(mappedBy="group")
private Set<Student> students;

@OneToMany(cascade=ALL)
private Set<Student> students;

@OneToMany(fetch = EAGER)
private Set<Student> students;
```

Анотація *@OneToOne* – якщо об'єкт А посилається тільки на один об'єкт В, а об'єкт В в свою чергу посилається тільки на один об'єкт А, тоді це відношення один-до-одного. Для вказування такого зв'язку з обох боків відношення використовується дана анотація. Можливі властивості даної анотації cascade, fetch, mappedBy, optional такі ж як і у попередніх анотацій. Інші властивості, можливі в даній анотації cascade і fetch, такі ж, як і для попередньої анотації.

```
@OneToOne (cascade=ALL)
private Address addr;

@OneToOne (fetch = LAZY)
private Address addr;

@OneToOne (mappedBy="user")
private Address addr;

@OneToOne (optional = false)
private Address addr;
```

Анотація *@ManyToMany* – якщо об'єкт А посилається на декілька об'єктів В, а кожний об'єкт В може посилатися на декілька об'єктів А, тоді це відношення багато-до-багатьох. Для вказування такого зв'язку з обох боків відношення використовується дана анотація. Можливі властивості анотації cascade, fetch, mappedBy такі ж як і у попередніх анотацій.

```
@ManyToMany (cascade=ALL)
private List<Student>;

@ManyToMany (fetch = LAZY)
private List<Student>;

@ManyToMany (mappedBy="st")
private List<Student>;
```

Анотація *@OrderBy* – за замовчуванням зв'язані списки вибираються з бази даних в не відсортованому вигляді. Для автоматичного сортування списку використовується дана анотація. Без параметрів за замовчуванням відбувається сортування по першому ключу. В параметрі даної анотації можна вказати назву поля по якому необхідно відсортувати, а також напрям сортування (ASK – зростаючий порядок, DESC – спадаючий порядок).

```
@OneToMany
@OrderBy()
private Set<Student> students;

@OneToMany
@OrderBy("fio ASC")
private Set<Student> students;
```

3.2.5 Управління сутностями

Об’єкт сутність може знаходитися в одному із трьох станів. *Transient object* – об’єкти в даному стані це заповнені екземпляри класів-сутностей. Вони можуть бути збережені в баз даних. Вони не приєднані до сесії, а поле id не повинно бути заповненим, бо інакше об’єкт має статус detached. *Persistent object* – об’єкт в даному стані – це збережена сутність, яка приєднана до певної сесії. Тільки в цьому стані об’єкт взаємодіє з базою даних. При роботі з об’єктом даного типу в рамках транзакції всі зміни об’єкта записуються в базу.

Detached object – об’єкт в даному стані це об’єкт , відєднаний від сесії, може існувати і не існувати в БД.

Будь-який об’єкт-сутність можна переводити із одного стану в інший. Для цього в інтерфейсі Session існують наступні методи:

persist(Object) – перетворює об’єкт з transient в persistent, тобто, приєднує до сесії і зберігає в БД. Однак, якщо ми присвоїмо значення полю id об’єкта, то отримаємо PersistentObjectException – Hibernate буде вважати, що об’єкт detached, тобто, існує в БД. При збереженні метод persist одразу виконує insert не роблячи select.

merge(Object) – перетворює об’єкт з transient чи detached в persistent. Якщо з transient, то працює аналогічно persist (генерує для об’єкта новий id, навіть якщо він зданий), якщо з detached – завантажує об’єкт з БД, приєднує до сесії, а при збереженні виконує запит update.

replicate(Object, ReplicationMode) – перетворює об’єкт з detached в persistent. При цьому у об’єкта обов’язково повинен бути завчасно встановлений id. Даний метод призначений для збереження в БД об’єкта з заданим id, чого не дозволяють зробити persist і merge. Якщо об’єкт з заданим id уже існує в БД, то поведінка визначається відповідно правилу з перерахування org.hibernate.ReplicationMode:

- ReplicationMode.IGNORE — нічого не міняється в базі;
- ReplicationMode.OVERWRITE — об’єкт зберігається в базу замість існуючого;
- ReplicationMode.LATEST_VERSION — в базі зберігається об’єкт з останньою версією;
- ReplicationMode.EXCEPTION — генерується виключення.

delete(Object) – видаляє об’єкт з БД, іншими словами, перетворює persistent в transient. Object може бути в будь-якому статусі, головне, щоб був встановлений id.

save(Object) – зберігає об’єкт в БД, генеруючи новий id, навіть, якщо він

встановлений. Object може бути в статусі transient чи detached.

update(Object) – оновлює об’єкт БД, перетворюючи його в persistent (Object в статусі detached).

saveOrUpdate(Object) — викликає save чи update.

refresh(Object) — оновлює detached-об’єкт, виконавши select до БД, і перетворює його в persistent.

get(Object.class, id) — отримує з БД об’єкт класу-сутності з певним id в статусі persistent.

Об’єкт Session кешує у себе завантажені об’єкти; при завантаженні об’єкта з БД в першу чергу перевіряється кеш. Для того, щоб видалити об’єкт з кеша і від’єднати від сесії, використовується метод session.evict(Object). Метод session.clear() застосовує evict до всіх об’єктів в сесії.

А тепер звернемо увагу на анотації @OneToMany и @ManyToOne в класах сутностях. Параметр fetch в @OneToMany позначає, коли завантажувати дочірні об’єкти. Може мати одне з двох значень, вказаних в перерахуванні javax.persistence.FetchType:

- FetchType.EAGER — завантажувати колекцію дочірніх об’єктів одразу при завантаженні батьківських об’єктів;
- FetchType.LAZY — завантажувати колекцію дочірніх об’єктів при першому зверненні до неї (виклик get) — так зване відкладене завантаження.

Параметр cascade позначає, які з методів інтерфейсу Session будуть поширюватися каскадно до асоційованих сутностей. Необхідно правильно налаштувати CascadeType для того, щоб не відвантажувати з бази багато лишніх асоційованих об’єктів-сутностей.

3.2.6 Виконання запитів

В Hibernate існує чотири типи запитів, з допомогою можна отримувати дані за бази даних:

- запити Native SQL;
- запити по критерію;
- запити про прикладу;
- запити HQL.

Розглянемо на прикладах кожен з типі запитів.

Запити NativeSQL є дуже гнучким типом запитів в Hibernate. Результатом запиту може бути не тільки колекція доменних об’єктів, але й колекція об’єктів представлених класом Map. Такі запити називаються скалярними. Приклад скалярного запити:

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
for(Object object : results) {
  Map row = (Map)object;
  System.out.print(row.get("first_name"));
```

```
  System.out.println(row.get("salary"));
}
```

Якщо результатом запиту буде колекція доменних об’єктів то запит буде виглядати наступним чином:

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

Приклад параметризованого запиту Native SQL приведений нижче:

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```

Дуже зручними типами запитів в Hibernate є запити по критерію і по прикладу. Їх можна використовувати, якщо не потрібно створювати дуже складний запит до декількох доменних об’єктів. Приклади запитів по критерію приведені нижче:

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));
// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));
// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));
// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));
// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));
// To check if the given property is null
cr.add(Restrictions.isNull("salary"));
// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));
// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));
// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));
// To sort records in descening order
cr.addOrder(Order.desc("salary"));
// To sort records in ascending order
cr.addOrder(Order.asc("salary"));
List results = cr.list();
```

Приклад запиту по прикладу приведений нижче:

```
Student student = new Student();
student.setFirstName("Tom");
Criteria criteria = session.createCriteria(Student.class);
```

```
criteria.add(Example.create(student)
  .enableLike()
  .excludeZeroes()
  .excludeProperty("sex")
  .ignoreCase()
);
List<Student> result = criteria.list();
```

Мова запиті HQL це мова, подібна до мови запитів SQL, але на відміну від SQL замість таблиці баз даних в запиті вказуються об’єкти Hibernate. З допомогою цієї мови можна писати запити не тільки на отримання даних, але і на їх зміну:

```
//FROM Clause
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();

//SELECT Clause
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();

//WHERE Clause
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();

//WHERE Clause
String hql = "FROM Employee E WHERE E.id > 10 " + "ORDER BY E.firstName DESC,
E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();

//GROUP BY Clause
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " + "GROUP BY
E.firstName";
Query query = session.createQuery(hql);
List results = query.list();

//UPDATE Clause
String hql = "UPDATE Employee set salary = :salary "
  + "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();

//DELETE Clause
String hql = "DELETE FROM Employee " + "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
```

```
int result = query.executeUpdate();

INSERT Clause
String hql = "INSERT INTO Employee(firstName, lastName, salary)" + "SELECT
firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
```

Параметризований запит з використанням мови HQL буде виглядати наступним чином:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

3.2.7 Спадкування в JPA

Специфікація JPA пропонує декілька стратегій для відображення ієрархії класі на таблиці бази даних:

1. MappedSuperclass – батьківські класи не можуть бути сутностями.
2. Single Table – сутності різних класів з спільним предком розміщуються в одній таблиці.
3. Joined Table – кожен підклас має свою окрему таблицю але властивості батьківського класу також знаходяться в окремій таблиці і для їх запиту необхідно дочірньому класу треба звертатися до батьківської таблиці
4. Table-Per-Class – кожен клас має свою окрему таблицю, в якій знаходяться всі властивості класу, включаючи батьківські властивості.

Кожна зі стратегій приводить до різних структур бази даних. Розглянемо більш детально кожну зі стратегій. При використанні стратегії MappedSuperclass спадкування проявляється тільки в класі, але не в моделі сутностей. Нехай у нас буде клас Person, який буде представляти батьківський клас:

```
@MappedSuperclass
public class Person {

  @Id
  private long personId;
  private String name;

  //constructor, getters, setters
}
```

Далі додамо клас Employee:

```
@Entity
public class MyEmployee extends Person {
  private String company;
 //constructor, getters, setters
}
```

В базі даних це буде відповідати тільки одній таблиці MyEmployee з трьома колонками для об'єднаних і успадкованих полів класу. Зверніть увагу, що класу Person немає анотації @Entity, оскільки він сам по собі не зберігається базу даних.

Стратегія Single Table створює одну таблицю для кожної ієрархії класів. Ми можемо визначити стратегію, яку хочемо використовувати, використавши анотацію @Inheritance в суперкласі. Ідентифікатор сутностей також визначений в суперкласі.

```
@Entity(name="products")
@Inheritance(strategy = InheritanceType.SINGLE__TABLE)
@DiscriminatorColumn(name="product__type",
discriminatorType = DiscriminatorType.INTEGER)
public class MyProduct {
   //...
}
```

Тоді ми можемо додати сутності підкласів:

```
@Entity
@DiscriminatorValue("1")
public class Book extends MyProduct {
//...
}

@Entity
@DiscriminatorValue("2")
public class Pen extends MyProduct {
//...
}
```

Оскільки записи для всіх сутностей будуть знаходитися в одній таблиці, то Hibernate потрібний спосіб, щоб їх розрізняти. За замовчуванням це робиться через колонку дискримінатора DTYPE, в якій в якості значення вказується ім'я об'єкта. Щоб налаштувати дискримінатор, треба використати анотацію @DiscriminatorColumn. Для того, щоб вказати яке значення буде мати дискримінатор для кожного об'єкту ієрархії треба використати анотацію @DiscriminatorValue.

При використанні стратегії Joined Table кожен клас в ієрархії співставляється з його таблицею. Єдина колонка, яка буде неодноразово з'являтися у всіх табличках – це ідентифікатор, який буде використовуватися для об'єднання сутностей при необхідності. Суперклас, який використовує цю стратегію виглядає наступним чином:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
  @Id
  private long animalId;
  private String species;

  //constructor, getters, setters
```

```
}
```

А підклас буде виглядати наступним чином:

```
@Entity
@PrimaryKeyJoinColumn(name = "petId")
public class Pet extends Animal {
  //...
}
```

Обидві таблиці будуть мати колонку animalId. Для налаштування первинного ключа дочірнього об’єкта використовується анотація @PrimaryKeyJoinColumn.

Недоліком такого методу відображення спадкування є те, що отримання об’єктів використовується об’єднання між таблицями, що може призвести до зниження продуктивності для великої кількості записів.

Стратегія Table-Per-Class відображає кожну сутність в її таблицю, яка містить всі властивості сутності, включаючи успадковані. Результуюча схема аналогічна тій, в якій використовується анотація @MappedSuperclass, з тою різницею, що для батьківського класу таблиця теж буде створюватися. Щоб використовувати цю стратегію нам треба тільки додати анотацію @Inheritance в базовий клас:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE__PER__CLASS)
public class Vehicle {
  @Id
  private long vehicleId;
  private String manufacturer;
  //standard constructor, getters, setters
}
```

Тоді ми можемо створити підкласти стандартним чином.

3.2.8 Модульне тестування програмного забезпечення

Даний розділ містить інформацію, що не пов’язана з технологією об’єктно-реляційного відображення, але буде використовуватися для тестування даної лабораторної роботи.

На сьогоднішній день дуже поширеною є техніка розробки програмного забезпечення test-driven development (TDD) при якій спочатку пишеться тест на певний функціонал, а тоді пишеться реалізація цього функціоналу. На практиці, звичайно, все не настільки ідеально, але в результаті код не тільки написаний і протестований, але тести неявно задають вимоги до функціоналу, а також показують приклад використання цього функціоналу. Одним за найвідоміших і найбільш використовуваним фреймворком для тестування є, мабуть, JUnit. Існує декілька його варіантів, наприклад, JUnit 3 і JUnit 4 JUnit 5. В якості прикладу будемо використовувати фреймворк JUnit 4. На відміну від JUnit 3 JUnit 4 побудований на анотаціях.

Анотація @Before позначає методи, які будуть викликані до виконання тесту. Методи повинні бути public void. В цих методах зазвичай розміщують налаштування для тесту, наприклад, генерацію тестових даних.

Анотація @BeforeClass позначає методи, які будуть викликані до створення екземпляру тестового класу. Методи повинні бути public static void. В даних методах слід розмішувати налаштування, загальні для декількох тестових методів класу.

Анотація @After позначає методи, які будуть викликані після виконання тесту. Методи повинні бути public void. В даному методі розміщуються операції звільнення ресурсів після тесту, наприклад, очистка тестових даних.

Анотація @AfterClass зв'язана по смислу з анотацією @BeforeClass, але виконує метод після виконання всіх тестів. Метод повинен бути public static void. Анотація @Test позначає тестові методи. Даний метод повинен бути public void. В методі, поміченому даною анотацією, розміщують тестові перевірки. Крім цього в даній анотації є два параметри. Параметр expected задає очікуване виключення і параметр timeout задає час, по закінченню якого тест вважається проваленим.

```
@Test(expected = NullPointerException.class)
public void testToHexStringWrong() {
  StringUtils.toHexString(null);
}

@Test(timeout = 1000)
public void infinity() {
  while (true);
}
```

Якщо який-небудь тест необхідно відключити (наприклад, цей тест постійно не проходить, але його виправлення відкладено на майбутнє), його можна помітити анотацією @Ignore. Якщо помітити даною анотацією клас то всі тести даного класу будуть вимкнені.

```
@Ignore
@Test(timeout = 1000)
public void infinity() {
  while (true);
}
```

Приклад тесту з використанням JUnit 4 приведений нижче:

```
public class StringUtilsJUnit4Test extends Assert {
  private final Map<String, byte[]> toHexStringData =
    new HashMap<String, byte[]>();

  @Before
  public static void setUpToHexStringData() {
    toHexStringData.put("", new byte[0]);
    toHexStringData.put("01020d112d7f", new byte[] { 1, 2, 13, 17, 45, 127 });
    toHexStringData.put("00fff21180", new byte[] { 0, -1, -14, 17, -128 });
    //...
  }

  @After
  public static void tearDownToHexStringData() {
```

```
    toHexStringData.clear();
  }

  @Test
  public void testToHexString() {
    for (Map.Entry<String, byte[]> entry : toHexStringData.entrySet()) {
      final byte[] testData = entry.getValue();
      final String expected = entry.getKey();
      final String actual = StringUtils.toHexString(testData);
      assertEquals(expected, actual);
    }
  }
}
```

3.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення набору класів для взаємодії з базою даних з використанням ORM Hibernate. В якості прикладу був обраний об’єкт предметної області «інтернет-магазин» – продукт. Результатом виконання всіх кроків, описаних нижче, не буде ні консольний ні графічний застосунок. Результатом буде набір класів, працездатність яких перевіряється з використанням інтеграційних тестів з використанням JUnit. Набір класів, отриманий в даній лабораторній роботі, буде використовуватися в усіх наступних лабораторних роботах. Тому до виконання даної лабораторної роботи треба поставитися особливо відповідально, оскільки вона може вплинути на всі наступні лабораторні роботи.

1. Встановіть в IDE Eclipse плагін MoreUnit. Він буде використовуватися для спрощення створення тестових класів.
2. Перевірте наявність СУБД PostgeSQL і системи автоматизованого збирання Gradle. Якщо СУБД PostgeSQL і системи автоматизованого збирання Gradle відсутні, встановіть їх, як показано в лабораторній роботі № 2.
3. Створіть новий Java проект.
4. В корінь проекту додайте build.gradle файл з контентом як показано нижче:

```
apply plugin: 'java'

repositories {
  mavenCentral()
}

sourceSets {
  main {
    java {
      srcDir 'src'
    }
  }
}
```

```
dependencies {
  runtime group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
  runtime group: 'org.hibernate', name: 'hibernate-core',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-entitymanager',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-osgi',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-envers',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-c3p0',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-proxool',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-infinispan',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-ehcache',
    version: '4.3.6.Final'
  runtime group: 'antlr', name: 'antlr', version: '2.7.6'
  runtime group: 'commons-collections', name: 'commons-collections',
    version: '3.1'
  runtime group: 'dom4j', name: 'dom4j', version: '1.6.1'
  runtime group: 'javassist', name: 'javassist', version: '3.4.GA'
  runtime group: 'javax.transaction', name: 'jta', version: '1.1'
  runtime group: 'org.slf4j', name: 'slf4j-api', version: '1.5.6'
  runtime group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.5.6'
}

jar {
  manifest {
    attributes 'Main-Class': 'ua.cn.stu.main.DatabaseExplorer'
  }
}

task copyToLib(type: Copy) {
  from configurations.runtime
  into "lib"
}

project.afterEvaluate {
  tasks.build.dependsOn(copyToLib)
}
```

5. В корені проекту створіть каталог lib.

6. Відкрийте термінал Windows, перейдіть в корінь проекту та запустіть команду gradle build, що спричинить компіляцію класів проекту, завантаження бібліотек з репозитарія Maven і копіювання бібліотек в каталог lib. Поновіть каталог lib. Додайте всі бібліотеки до Java Build Path (Build Path -> Configure Build Path -> Libraries -> Add JARs).

7. В каталозі src створіть пакет ua.cn.stu.domain.
8. Додайте в пакет ua.cn.stu.domain клас доменного об’єкта:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "producttable")
public class Product {

  public Product() {
  }

  public Product(String name, String description) {
    this.name = name;
    this.description = description;
  }

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Long id;

  @Column(name = "productname")
  private String name;

  @Column(name = "productdescription")
  private String description;

  public Long getId() {
    return id;
  }

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

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getDescription() {
    return description;
  }
```

```
  public void setDescription(String description) {
    this.description = description;
  }
}
```

9. Створіть в каталозі src пакет ua.cn.stu.dao.

10. В пакеті ua.cn.stu.dao створіть клас-фабрику з ім’ям HibernateDAOFactory. Він буде використовуватися для створення об’єкту SessionFactory, для відкриття сесії Hibernate і для зберігання об’єктів DAO для різних доменних об’єктів. В даному прикладі HibernateDAOFactory зберігає тільки один об’єкт DAO – ProductDAO. При доданні нових доменних об’єктів для кожного нового об’єкта необхідно створити новий клас DAO.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.cfg.Environment;
import org.hibernate.service.ServiceRegistry;

import ua.cn.stu.domain.Product;

public class HibernateDAOFactory {

  private static HibernateDAOFactory instance;

  private ProductDAO productDAO;

  private Session session;

  //Ініціалізація синглетону
  public static HibernateDAOFactory getInstance() {
    if (null == instance) {
      instance = new HibernateDAOFactory();
    }
    return instance;
  }

  //Створення об’єкта Session для взаємодії з Hibernate
  private Session getSession() {
    if (null == session) {
      Configuration configuration = new Configuration();
      configuration.setProperty(Environment.DRIVER, "org.postgresql.Driver");
      configuration.setProperty(Environment.URL,
        "jdbc:postgresql://localhost:5432/shop");
      configuration.setProperty(Environment.USER, "postgres");
      configuration.setProperty(Environment.PASS, "111111");
      configuration.setProperty(Environment.DIALECT,
        "org.hibernate.dialect.PostgreSQLDialect");
      configuration.setProperty(Environment.HBM2DDL_AUTO, "create");
```

```
      configuration.setProperty(Environment.SHOW_SQL, "true");
      configuration.addAnnotatedClass(Product.class);
      StandardServiceRegistryBuilder serviceRegistryBuilder =
        new StandardServiceRegistryBuilder();
      serviceRegistryBuilder.applySettings(configuration.getProperties());
      ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
      SessionFactory sessionFactory =
        configuration.buildSessionFactory(serviceRegistry);
      session = sessionFactory.openSession();
    }
    return session;
  }

  public ProductDAO getProductDAO() {
    if (null == productDAO) {
      productDAO = new ProductDAO(getSession());
    }
    return productDAO;
  }
}
```

11. В пакеті ua.cn.stu.dao створіть клас ProductDAO, що буде містити набір методів для додання, видалення і заміни доменного об’єкту Product. Більшість методів даного класу будуть однаковими для будь якого доменного об’єкта, тому, якщо кількість доменних об’єктів в системі збільшиться, то більшість методів класу ProductDAO необхідно винести в шаблонний клас.

```
import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.SQLQuery;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.criterion.Restrictions;

import ua.cn.stu.domain.Product;

public class ProductDAO {

  private Session session;

  public ProductDAO(Session session) {
    this.session = session;
  }

  /**
   * This method create new entity
   */
  public Product createProduct(Product product) {
    Transaction transaction = session.beginTransaction();
    session.saveOrUpdate(product);
    transaction.commit();
```

```
  return product;
}

/**
 * This method update existing product
 */
public Product updateProduct(Product product) {
  Transaction transaction = session.beginTransaction();
  session.merge(product);
  transaction.commit();
  return product;
}

/**
 * This method delete existing product
 */
public void deleteProduct(Product product) {
  Transaction transaction = session.beginTransaction();
  session.delete(product);
  transaction.commit();
}

/**
 * This method remove entity by id
 */
public void deleteProductById(Long productId) {
  Product product = (Product) session.get(Product.class, productId);
  deleteProduct(product);
}

/**
 * This method return all entities
 */
public List<Product> getAllProducts() {
  SQLQuery query = session.createSQLQuery(
    "select * from producttable").addEntity(Product.class);
  List<Product> productList = query.list();
  return productList;
}

/**
 * This method return all products by name
 * This method is specific for Product domain object
 *instead of other methods from this class
 * All other methods can be moved to generic class and
 *can be used for other domain objects.
 */
public List<Product> getProductsByName(String name) {
  Criteria criteria = session.createCriteria(Product.class)
    .add(Restrictions.eq("name", name));
```

```
    return criteria.list();
  }
}
```

12. Створіть source каталог з ім’ям test.

13. Для виконання наступного кроку в IDE Eclips має бути встановлений і налаштований плагін MoreUnit. Натисніть комбінацію клавіш Ctrl+U. В діалоговому вікні, що відкриється введіть ім’я класу ProductDAOTest та оберіть тестовий фреймворк JUnit 5 (див. рисунок 3.3). В результаті в каталозі test повинен створитися пакет ua.cn.stu.dao а в ньому клас ProductDAOTest, який ми будемо використовувати для перевірки працездатності нашого набору класів.

Рисунок 3.3 – Вікно створення тестового класу

14. Додайте бібліотеку JUnit 5 до проекту. Натисніть правою кнопкою миші на проект і в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні перейдіть на вкладку Libraries та натисніть кнопку Add Library. В діалоговому вікні оберіть JUnit, в наступному діалоговому вікні оберіть версію JUnit 5 та натисніть кнопку Finish.

15. Додайте до класу ProductDAOTest набір методів для тестування функціоналу створених класів. Набір тестів даного класу не є повним, але достатнім для демонстрації можливостей JUnit 5. В умовах реального проекту тестів буде набагато більше.

```
import static org.junit.Assert.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;

import java.util.List;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
import ua.cn.stu.domain.Product;

public class ProductDAOTest {

  private Product product1;
  private Product product2;
  private Product product3;

  @BeforeEach
  public void before() {
    // Create test products before starting test
    product1 = new Product("Test product name 1",
      "Test product description 1");
    product2 = new Product("Test product name 2",
      "Test product description 2");
    product3 = new Product("Test product name 3",
      "Test product description 3");
    HibernateDAOFactory.getInstance().getProductDAO().createProduct(product1);
    HibernateDAOFactory.getInstance().getProductDAO().createProduct(product2);
    HibernateDAOFactory.getInstance().getProductDAO().createProduct(product3);
  }

  @AfterEach
  public void after() {
    // Remove all products after completing test
    List<Product> productList = HibernateDAOFactory.getInstance()
      .getProductDAO().getAllProducts();
    for (Product product : productList) {
      HibernateDAOFactory.getInstance().getProductDAO().deleteProduct(product);
    }
  }

  @Test
  public void whenGetAllThenAllShouldBePresent() {
    // given
    // when
    List<Product> productList = HibernateDAOFactory.getInstance()
      .getProductDAO().getAllProducts();
    // then
    assertEquals(3, productList.size());
    for (Product product : productList) {
      assertNotNull(product);
      assertNotNull(product.getId());
      assertNotNull(product.getName());
      assertNotNull(product.getDescription());
    }
  }

  @Test
  public void whenAddProductThenItShoudBePresentInDb() throws Exception {
```

```
  // given
  Product product = new Product("Test product name",
    "Test product description");
  // when
  HibernateDAOFactory.getInstance().getProductDAO().createProduct(product);
  // then
  List<Product> productListAfter = HibernateDAOFactory.getInstance()
    .getProductDAO().getAllProducts();
  int afterSize = productListAfter.size();
  assertNotNull(product.getId());
  assertEquals(4, afterSize);
}

@Test
public void whenDeleteProductThenItShoudNotBePresentInDb() throws Exception {
  // given
  Product product = new Product("Test product name",
    "Test product description");
  HibernateDAOFactory.getInstance().getProductDAO().createProduct(product);
  // when
  HibernateDAOFactory.getInstance().getProductDAO().deleteProduct(product);
  // then
  List<Product> productListAfter = HibernateDAOFactory.getInstance()
    .getProductDAO().getAllProducts();
  assertEquals(3, productListAfter.size());
  for (Product productEnt : productListAfter) {
    assertNotEquals("Test product name", productEnt.getName());
    assertNotEquals("Test product description", productEnt.getDescription());
  }
}

@Test
public void whenGetAllProductsByNameThenAllWithSuchNameShouldBePresent() {
  //given
  Product product1 = new Product("Specific product name",
    "Test product description1");
  Product product2 = new Product("Specific product name",
    "Test product description2");
  Product product3 = new Product("Specific product name",
    "Test product description3");
  HibernateDAOFactory.getInstance().getProductDAO().createProduct(product1);
  HibernateDAOFactory.getInstance().getProductDAO().createProduct(product2);
  HibernateDAOFactory.getInstance().getProductDAO().createProduct(product3);
  //when
  List<Product> productListByName = HibernateDAOFactory.getInstance()
    .getProductDAO().getProductsByName("Specific product name");
  //then
  assertEquals(3, productListByName.size());
  for (Product productEnt : productListByName) {
    assertEquals("Specific product name", productEnt.getName());
```

```
        }
    }
}
```

16. Натисніть правою кнопкою миші на клас ProductDAOTest та в контекстному меню натисніть Run As - > JUnit Test.

17. Переконайтеся, що всі тести пройшли успішно. Якщо тести пройшли успішно, то всі тести будуть пофарбовані в зелений колір, як показано на рисунку 3.4.

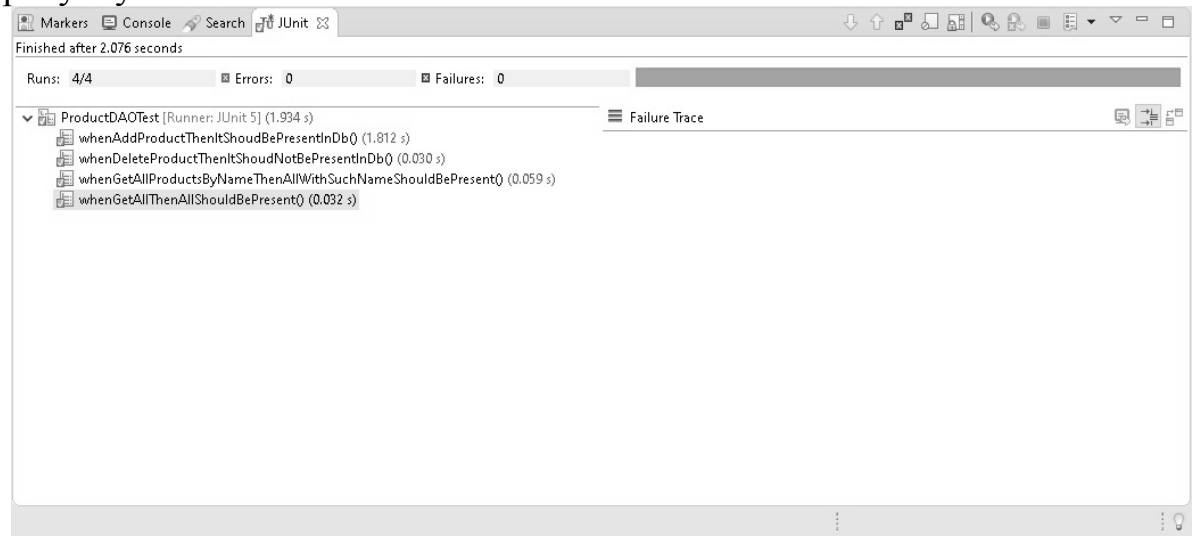

Рисунок 3.4 – Вікно успішного проходження тестів

18. Вдоскональте застосунок відповідно до завдання на лабораторну роботу.

3.4 Завдання на лабораторну роботу

Вдосконалити застосунок, описаний в ході виконання лабораторної роботи, щоб він міг працювати з двома сутностями, на відміну від застосунку, описаного в лабораторній роботі. Предметну область необхідно вибрати з таблиці 3.1. Номер варіанта визначається по останній цифрі залікової книжки. Сутності повинні бути пов’язані відношенням один до багатьох.

Таблиця 3.1. Варіанти завдань

Номер варіанту	Предметна область
1	Піцерія (Офіціант, Відвідувач)
2	Крамниця (Постачальник, Товар)
3	Супермаркет (Продавець, Товар)
4	Відділення поліції (Поліцейський, Порушник)
5	Лікарня (Лікар, Пацієнт)
6	Кінотеатр (Кінозал, Відвідувач)
7	Корабель (Каюта, Пасажир)
8	Підприємство (Відділ, Співробітник)
9	Бібліотека (Книга, Читач)

3.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

3.6 Контрольні питання

1. Що таке об’єктно-реляційне відображення;
2. Причини і переваги використання ORM;
3. Назвіть основні компоненти архітектури Hibernate і для чого вони слугують;
4. Назвіть обмеження, що накладаються на persistent об’єкти.
5. Як вказати первинний ключ і його автоматичну генерацію з допомогою анотацій;
6. Які стратегії спадкування існують в Hibernate;
7. Які види запитів є в Hibernate;
8. Який параметр конфігурації Hibernate відповідає за створення схеми бази даних.
9. Розкажіть, як виконати запит для отримання об’єктів.
10. Чим параметризовані запити HQL відрізняються від параметризованих запитів SQL з використанням JDBC.

3.7 Завдання до захисту лабораторної роботи

1. Написати метод на мові Java, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів Native SQL.
2. Написати метод на мові Java, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів по критерію.
3. Написати метод на мові Java, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів за прикладом.
4. Написати метод на мові Java, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів HQL.
5. Винести ідентифікатор об’єкта (поле id) в клас BaseEntity. Зробити клас BaseEntity базовим для обох доменних об’єктів за варіантом. Необхідно використати стратегію спадкування Single Table Strategy.
6. Винести ідентифікатор об’єкта (поле id) в клас BaseEntity. Зробити клас BaseEntity базовим для обох доменних об’єктів за варіантом. Необхідно використати стратегію спадкування Table Per Class Strategy.
7. Винести ідентифікатор об’єкта (поле id) в клас BaseEntity. Зробити клас BaseEntity базовим для обох доменних об’єктів за варіантом. Необхідно використати стратегію спадкування Joined Strategy.

8. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати мову запитів HQL.
9. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати запити по критерію.
10. Замінити базу даних, використану в лабораторній роботі, на вбудовану базу даних, наприклад Derby або H2.
11. Налаштувати кешування доменних об’єктів з використанням вбудованих механізмів Hibernate, або додаткових бібліотек таким чином, щоб об’єкти зі сторони один кешувались а об’єкти зі сторони багато не кешувались.

4 ЛАБОРАТОРНА РОБОТА №4
ВИВЧЕННЯ JAVA ТЕХНОЛОГІЙ SERVLET І JSP

4.1 Мета роботи

Вивчити Java технології Servlet і JSP для реалізації інтерфейсу користувача.

4.2 Теоретичні відомості

4.2.1 Технологія Servlet

Сервлети – це особливим чином написані (відповідно специфікації) Java класи, які виконуються віддалено на сервері і виклик яких здійснюється віддалено з web-браузера з допомогою HTTP протоколу через web-сервер.

Після появи технології сервлетів розробники зіткнулися з однією великою незручністю – для генерації HTML сторінки на льоту з допомогою сервлета весь HTML код треба було розміщати в самому сервлеті. Виходило, що HTML код сторінки (presentation) змішувався з java кодом (logic), що утруднювало роботу як програміста, так і дизайнера. Для вирішення цієї проблеми була винайдена технологія JSP – Java Server Pages. Вона нагадувала існуючі на той момент технології ASP (від Microsoft) і ColdFusion (від Allaire). Але схожість була чисто зовнішньою. Так само як і в ASP і в ColdFusion в JSP можна вставляти java код в HTML код сторінки, але якщо в ASP і в ColdFusion цей код при кожному виклику інтерпретувався, то в JSP при першому виклику JSP сторінки цей код приховано для розробника транслюється в сервлет і компілюється, після чого при наступному зверненні до web-сервера викликається вже не транслятор самої JSP сторінки, а відкомпільований сервлет. Зрозуміло, що при внесенні змін в JSP сторінку web-сервер виявляє, що сторінка змінилася і знову оновлює сервлет, що відповідає даній сторінці.

В технологіях Servlet і JSP введено визначення контейнера (container). Якщо коротко, то Servlet-контейнер – це програма, що відповідає за виконання сервлетів. JSP-контейнер - це програма, що відповідає за трансляцію JSP сторінок в сервлети і передачу цих сервлетів Servlet-контейнеру. Так як сервлети і JSP сторінки викликаються через протокол HTTP, то сервлет контейнер і JSP-контейнер часто супроводжує ще один компонент – web-сервер, який також може бути написаний на мові Java. Web-сервер отримує запити від браузера на виконання того чи іншого сервлета чи JSP сторінки на сервері, передає запит в контейнер, який виконує той чи інший сервлет. Результати виконання повертаються контейнером web-серверу, який в свою чергу пересилає його в до браузера.

Розглянемо життєвий цикл сервлета. Архітектурно всі сервлети мають реалізовувати інтерфейс Servlet. Цей інтерфейс визначає 5 методів:

void init(ServletConfig config) – цей метод автоматично викликається один раз при виконанні сервлета для його ініціалізації. Параметр ServletConfig надається контейнером сервлетів, який виконує сервлет.

ServletConfig getServletConfig() – цей метод повертає посилання на об’єкт, який реалізує інтерфейс ServletConfig. Він надає доступ до конфігурації сервлета.

String getServletInfo() – даний метод повертає рядок з даними про сервлет.

void service(ServletRequest request, ServlerResponse response) – контейнер сервлетів викликає цей метод для відповіді на клієнтський запит.

void destroy() – цей метод викликається після завершення виконання сервлета контейнером сервлетів. Даний метод слід застосовувати для звільнення ресурсів, що використовуються сервлетом, таких як файли і відкриті з’єднання з БД. Загальна схема обробки клієнтських запитів зображена на рисунку 4.1.

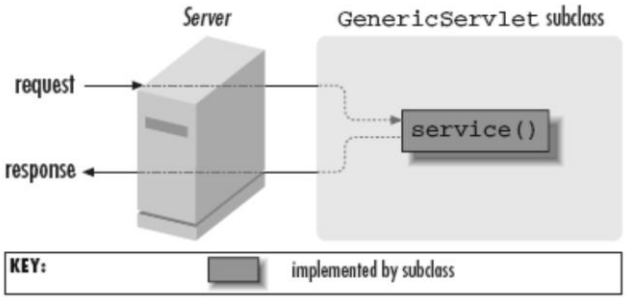

Рисунок 4.1 – Обробка клієнтських запитів

Сервлети, що працюють з клієнтом через web зазвичай розширюють клас HttpServlet. Метод service зазвичай перевизначається, щоб мати можливість розрізняти стандартні методи запитів, що отримуються від web-браузера клієнта. Типами запитів HTTP є get, post, delete, option, put, trace. Найбільш поширеними є get (отримання HTML документу) і post (відправка інформації на сервер). В класі HttpServlet визначені методи doGet і doPost для обробки реакції на ці запити. (рисунок 4.2).

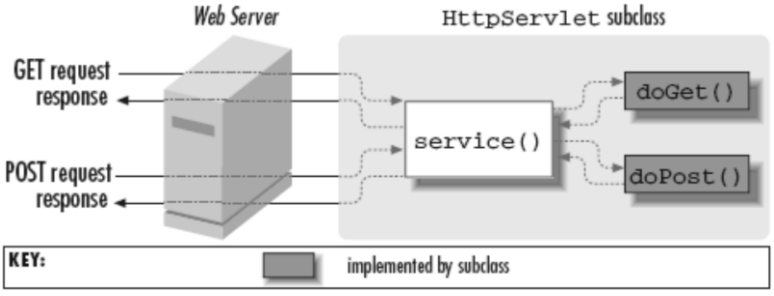

Рисунок 4.2 – Обробка запитів get і post классом HTPPServlet

Фільтри – це класи, які можуть перевизначать запит або змінювати відповідь. Він є своєрідним препроцесором запиту до сервлету і постпроцесором відповіді, що відправляється сервлетом. Фільтри мають наступні можливості:

- перехоплення звернення до сервлету в момент його виклику;
- перевірка запиту перед викликом сервлета;
- зміна заголовку запиту і даних запиту з наданням зміненої версії об’єкта запиту, який приховує реальний запит;

- зміна заголовка відповіді і даних відповіді з наданням зміненої версії об’єкта відповіді, який приховує реальну відповідь;
- перехоплення звернення до сервлету після його виклику.

Фільтр імплементує інтерфейс Filter і повинен реалізовувати три методи:

void init(FilterConfig config) – цей метод призначає фільтру конфігураційний об’єкт.

void destroy() – це метод виконується при завершенні роботи фільтра.

void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) – цей метод виконує конкретні дії фільтру.

Слухачі – класи, які можуть слідкувати за тим, коли контекст, сесія, запит були ініційовані, або відстежувати час, коли вони повинні бути знищені і коли атрибути бути додані або видалені з контексту або сесії. Існує декілька інтерфейсів, які повинні реалізовувати класи для того, щоб відслідковувати вищезазначені події: ServletContextListener, ServletContextAttributeListener, ServletRequestListener, ServletRequestAttributeListener. Для того, щоб реалізувати слухача, необхідно реалізувати один з вищеперерахованих інтерфейсів і додати конфігурацію в файл web.xml для реєстрації класу-слухача.

4.2.2 Технологія JSP

Технологія JSP сторінок (Java Server Pages) дозволяє без зусиль створювати web-контент, у якого є як статичний, так і динамічний компоненти. Основними характеристиками JSP технології є:

- мова розробки JSP сторінок, що є текстовими документами, які описують процес обробки запитів і конструювання відповіді;
- конструкції для отримання доступу до об’єктів зі сторони сервера;
- механізми, що визначають розширення для мови JSP.

JSP сторінкою є документ з текстовою основою, що містить два типи тексту: статичні шаблонні дані, що висловлені з допомогою будь-якого елементів, що створюють динамічний вміст.

JSP сторінка обслуговує запити так само, як це робить сервлет. Таким чином, життєвий цикл і більшість можливостей JSP сторінки (зокрема, динамічні аспекти) визначаються технологією Java Servlet. Коли запит відображається на сторінку JSP, він обробляється спеціальним сервлетом, який спочатку перевіряє, чи не є сервлет сторінки JSP старшим ніж сама сторінка JSP. Якщо це так, то він транслює сторінку JSP в клас сервлета і компілює клас. Однією з переваг процесу розробки сторінок JSP в порівнянні з розробкою сервлетів є те, що процес створення сторінок JSP здійснюється автоматично.

Під час фази трансляції кожний тип даних в сторінці JSP оброблюється окремо. Шаблонні дані перетворюються в програмний код, що буде відправляти дані в потік, що повертає дані клієнту. JSP елементи оброблюються наступним чином:

- директиви використовуються для управління web-контейнером в процесі трансляції і виконання їм JSP сторінки;

- елементи сценаріїв вставляються в клас сервлета сторінки JSP;
- елементи у вигляді <jsp:XXX ... /> конвертуються в виклики методів компонентів JavaBean або виклики Java Servlet API.

Для сторінки JSP з іменем pageName, джерело сервлета сторінки JSP зберігається в файлі:

```
<JWSDP_HOME>/work/StandardEngine/localhost/context_root/pageName$jsp.java
```

Обидві фази, як трансляція, так і компіляція, можуть викликати помилки, які відстежуються тільки після першого запиті до сторінки. Якщо помилка зустрічається під час трансляції сторінки (наприклад, в випадку, коли транслятор зустрічає погано сформований елемент JSP), сервер повертає виключення ParseException, а вихідний файл класу сервлета буде пустим або неповним. Останній неповний рядок вкаже на неправильний елемент JSP.

Якщо помилка зустрічається під час компіляції сторінки JSP (наприклад, в випадку синтаксичної помилки в скриплеті), сервер поверне виключення JasperException і повідомлення, що включає ім’я сервлету JSP сторінки з рядком, де була знайдена помилка. Якщо сторінка була трансльована і відкомпільована, сервлет сторінки JSP притримується життєвого циклу сервлету. Якщо екземпляр сервлета сторінки JSP не існує, контейнер завантажує клас сторінки JSP, створює екземпляр класу сервлета, ініціює клас сервлету шляхом виклику методу jspInit. Контейнер викликає метод jspService, передаючи об’єкт запиту і відповіді. Якщо контейнер потребує видалити сервлет сторінки JSP, він викликає метод jspDestroy.

Можна контролювати різні параметри виконання сторінок JSP з використання директиви page.

При виконанні сторінки JSP результат, записаний в об’єкт відповіді,автоматично заноситься в буфер. Можна встановлювати розмір буфера з допомогою наступної директиви page:

```
<%@ page buffer="none|xxxkb" %>
```

Більший розмір буфера дозволяє записати більший об’єм вмісту. Такими чином, сторінці JSP забезпечується більше часу для встановлення відповідних кодів і заголовків статусу, або для відправки іншому web-ресурсу. Менший розмір буферу зменшує завантаження пам’яті сервера і дозволяє клієнту отримати дані швидше.

Під час виконання сторінки JSP може виникнути будь-яка кількість виключних ситуацій. Можна задати, щоб web-контейнер у випадку виникнення виключної ситуації передавав управління сторінці помилок. Для цього необхідно включити наступну директиву page на початку сторінки JSP:

```
<%@ page errorPage="file_name" %>
```

Наприклад, можна вказати наступну директиву:

```
<%@ page errorPage="errorpage.jsp"%>
```

На початку сторінки з допомогою наступної директиви page вказано, що errorpage.jsp є сторінкою помилок:

```
<%@ page isErrorPage="true|false" %>
```

Дана директива робить доступним для сторінки помилок об'єкт виключення (типу javax.servlet.jsp.JspException) таким чином, щоб розробник міг отримувати, інтерпретувати і, можливо, навіть відображати дані про причину виникнення виключної ситуації на сторінці помилок. Можна також визначати сторінки помилок для WAR-файлу, який містить сторінку JSP. Якщо сторінки помилок визначені як для WAR так і для сторінки JSP, спочатку з'являється сторінка помилок сторінки JSP.

Модна налаштувати процес ініціалізації шляхом ер визначення методу jspInit інтерфейсу. Це дозволить сторінці JSP зчитувати постійні дані конфігурації, ініціювати ресурси і здійснювати будь-яку іншу одноразову дію. Звільняють ресурси, використовуючи метод jspDestroy. Зважаючи на те, що корпоративний компонент використовується спільно всіма сторінками JSP, його необхідно ініціювати при завантаженні застосунку, а не в кожній окремій JSP сторінці. Технологія Java Servlet забезпечує для цього події життєвого циклу застосунку і класи слухачі.

Статичний вміст створюється в сторінці JSP так само, як якби це була звичайна сторінка, що містить тільки текстовий формат даних. Статичний вміст може бути виражений в будь-якому форматі, наприклад HTML, WML, чи XML. Форматом визначеним за замовчуванням є HTML. За бажанням можна використовувати і інший формат тексту. Для цього необхідно включити на початку сторінки JSP директиву page з атрибутом contentType, якому в якості значення слід встановити тип формату. Наприклад, якщо необхідно, щоб дані були виражені в форматі WML (wireless markup language – мова розмітки для безпровідних систем), необхідно включити наступну директиву:

```
<%@ page contentType="text/vnd.wap.wml"%>
```

Динамічний вміст створюється шляхом звернення до об'єктів мови програмування Java і елементі сценаріїв.

4.2.3 Використання об'єктів в JSP сторінках

З сторінки JSP можна отримати доступ до різноманітних об'єктів, включаючи корпоративні компоненти і компоненти JavaBean. Технологія JSP автоматично організує доступ до певних об'єктів так що можна створювати об'єкти застосунку і звертатися до них з JSP сторінки.

Перш за все існують так звані неявні об'єкти, що створюються web-контейнером і містять дані, що відносяться до певного запиту, сторінки чи застосунку. Багато з таких об'єктів визначаються технологією Java Servlet, що лежить в основі технології JSP. В таблиці 4.1 представлені неявні об'єкти.

Таблиця 4.1 – Неявні об'єкти

Змінні	Клас	Пояснення
1	2	3
application	javax.servlet.ServletContext	Контекст для сервлета JSP сторінки і будь-якого web компонента, що міститься в тому ж самому застосунку.

Змінні	Клас	Пояснення
1	2	3
config	javax.servlet.ServletConfig	Інформація про ініціалізацію сервлета JSP сторінки.
exception	java.lang.Throwable	Доступно тільки з сторінки помилок (error page).
out	javax.servlet.jsp.JspWriter	Вихідний потік.
page	java.lang.Object	Екземпляр сервлета JSP сторінки, що оброблює поточний запит. Зазвичай не використовується авторами JSP сторінок.
pageContext	javax.servlet.jsp.PageContext	Контекст JSP сторінки. Містить єдиний API для управління різними контекстними атрибутами, що описані в Sharing Information. Цей API широко використовується, коли реалізують tag handlers (програма обробки тегів)
request	підтип javax.servlet.ServletRequest	Запит, що запускає виконання JSP сторінки.
response	підтип javax.servlet.ServletResponse	Відповідь, що повертається клієнту. Зазвичай не використовується авторами JSP сторінок.
session	javax.servlet.http.HttpSession	Об’єкт – сесія клієнта.

По можливості, логіку роботи застосунку бажано виносити в об’єкти так, щоб розробники сторінки моли сфокусуватися на аспектах відображення.

Існують чотири способи для створення і використання об’єктів в сторінках JSP:

- змінні екземпляру класу, що відносяться до класу сервлета сторінки JSP і створюються в оголошеннях і доступні в скриплетах і виразах;
- локальні змінні класу сервлету сторінки JSP створюються і використовуються в скриплетах і виразах;
- атрибути об’єктів області дії, що створюються і використовуються в скриплетах і виразах;
- компоненти JavaBean можуть створюватися і бути доступними з допомогою модернізованих елементів JSP.

Зі сторінки JSP, що завантажуються як багатопотокові сервлети, можна отримати доступ до об’єктів, до яких можна застосувати дані умови. Можна вказати web-контейнеру, яким чином слід відправляти багатопотокові запити клієнтів, використовуючи директиву page:

```
<%@ page isThreadSafe="true|false" %>
```

Коли значенням isThreadSafe є true, web-контейнер встановлює більш високий пріоритет відправки багатопотокових паралельних клієнтських запитів до сторінки JSP. Це значення використовується за замовчуванням. Використовуючи true, необхідно гарантувати, що відповідним чином був синхронізований доступ до всіх загальних об'єктів, визначним на рівні сторінки. Цей рівень включає в себе об'єкти, створені в оголошеннях компонентів JavaBean зі сторінкою в якості області дії і атрибути об'єкта області дії page. Якщо значення isThreadSafe є false, запити відправляються почергово (тобто, по одному) в тому порядку, в якому вони були отримані, а контролювати доступ до об'єктів сторінки не потрібно. Але все одно необхідно гарантувати вірну синхронізацію доступу до атрибутів об'єктів обрості дії application або session, а також до компонентів JavaBean з застосунком, або сесією, вказаними в якості області дії.

JSP елементи сценаріїв використовуються для створення об'єктів організації до них доступу, визначення методів і управління потоком контролю. Зважаючи на те, що однією з цілей технології JSP є відділення статичних шаблонних даних від програмного коду, що необхідний для динамічної генерації вмісту, рекомендовано помірне використання сценаріїв JSP. Більшу частину логіки, яка потребує використання сценаріїв можна реалізувати з використанням тегів користувача, або бібліотеки JSTL.

Технологія JSP дозволяє контейнеру підтримувати будь-яку мову створення сценаріїв, що має можливість викликати java об'єкти. Якщо замість мови java, що встановлена за замовчуванням, для створення сценаріїв є бажання використовувати іншу мову, її необхідно задати в директиві page на початку сторінки JSP:

```
<%@ page language="scripting language" %>
```

Через те, що елементи сценаріїв конвертуються в вирази мови програмування в класі сервлета сторінки JSP, на сторінку необхідно імпортувати всі класи і пакети, що використовуються даною сторінкою JSP.

Якщо сторінка написана на мові java, імпортований клас або пакет з допомогою директиви page виглядає наступним чином:

```
<%@ page import="packagename.*, fully_qualified_classname" %>
```

Наприклад:

```
<%@ page import="java.util.*, stu.jtap2.labs.lab2.domain.*" %>
```

JSP-оголошення використовується для оголошення змінних і методів на мові створення сценаріїв сторінки. Синтаксис для оголошень наступний:

```
<%! scripting language declaration %>
```

Коли мовою створення сценарію є java, змінні і методи в JSP-оголошеннях стають оголошеннями класу сервлету сторінки JSP.

Наприклад:

```
<%!
  private StudentManager stm;
  public void jspInit() {
    ...
```

```
  }
  public void jspDestroy() {
    ...
  }
%>
```

JSP-скриплет використовується для зберігання будь-якого фрагменту коду дійсного для мови сценаріїв, що використовується в даній сторінці. Синтаксис для скриплета виглядає наступним чином:

```
<%
scripting language statements
%>
```

Коли мовою сценаріїв є java, скриплет трансформується у фрагмент виразу мови java і вставляється в метод service сервлета сторінки JSP. Змінна мови програмування, створена в скриплеті, доступна з будь-якої точки сторінки JSP. Наступний приклад містить скриплет, що отримує ітератор зі списку студентів академічної групи, а також скриплет, що встановлює конструктор для циклічного проходу по всім елементам списку. Під час циклу сторінка JSP отримує властивості об’єктів і формує їх, використовуючи при цьому розмітку HTML. Блок відкриває цикл while, тому розмітка HTML йде перед скриплетом, що закриває блок.

```
<%
  Iterator i = group.getItems().iterator();
  while (i.hasNext()) {
    Student item = (Student) i.next();
%>
  <tr>
    <td align="right" bgcolor="#ffffff">
      <%=item.getFirstName()%>
    </td>
    <td align="right" bgcolor="#ffffff">
      <%=item.getFirstName()%>
    </td>
  </tr>
<%
  }
%>
```

JSP-вирази використовуються для вставки в потік даних, що повертається клієнту, значень виразів мови сценаріїв, конвертованого в рядок.

Коли мовою написання сценарію є java, вираз перетворюється в оператор, що конвертує значення виразу в об’єкт String і вставляє його в неявний об’єкт out. Синтаксис такого виразу виглядає наступним чином:

```
<%= scripting language expression %>
```

Необхідно звернути увагу, що використання крапки з комою в JSP-виразах заборонено, навіть в тих випадках, коли у такого ж виразу в скриплеті крапка з комою присутня. Наступний скриплет виводить кількість студентів академічної групи:

```
<%
```

```
  int num = group.getNumberOfItems();
  if (num> 0) {
%>
  <font size="+2">
    В группе <%=num%>
    <%=(num==1 ? "Студент" : "Студентов")%> <br/>
  </font>
<%
  }
%>
```

Існує два механізми для включення іншого web-ресурсу в сторінку JSP: використання директиви include і елементу jsp:include. Директива include оброблюється під час трансляції сторінки JSP в класі сервлета. Дією даної директиви є вставка тексту, що міститься в іншому файлі (або в статичному ресурсі, або ж в іншій сторінці JSP) в необхідну сторінку JSP. Зазвичай директива include використовується для включення і іншу сторінку контенту шапки, авторських даних або будь якої іншої порції даних, яку використовують багаторазово. Синтаксис для директиви include наступний:

```
<%@ include file="filename" %>
```

У даного підходу нема обмежень, так як директива include статично додає до кожного файлу ресурс, що використовується повторно, на який посилається директива.

Елемент jsp:include оброблюється під час виконання сторінки JSP. Цей елемент дозволяє включити в JSP файл як статичні так і динамічні ресурси. Результати, отримані при доданні статичних і динамічних ресурсів, будуть дещо відрізнятися. Якщо ресурс статичний, його вміст вставляється в файл JSP, що його викликає. Якщо ресурс є динамічним, то на ресурс відправляється запит. Виконується сторінка, що була додана, а тоді результат додається в відповідь JSP сторінки, яка викликає. Синтаксис елемента jsp:include виглядає наступним чином:

```
<jsp:include page="includedPage" />
```

Сервер Tomcat не буде перезавантажувати статично включену сторінку, яка була змінена, до тих пір, поки, сторінка, що її включає, також не буде змінена.

Механізм передачі управління іншому компоненту з сторінки JSP використовує виконувані функції, що забезпечені Java Servlet API. Доступ до цих функцій здійснюється з сторінки JSP з допомогою елементу jsp:forward:

```
<jsp:forward page="/main.jsp" />
```

Якщо клієнту вже повернули які-небудь дані, у елемента jsp:forward виникне виключна ситуація IllegalStateException.

При викликанні елементу include або forward цільовій сторінці надається початковий об’єкт запиту. Якщо необхідно помістити в цю сторінку додаткові дані, з допомогою об’єкта jsp:param можна додати до обєкта запиту і інші параметри:

```
<jsp:include page="header.jsp">
<jsp:param name="param1" value="value1"/>
```

```
</jsp:include>
```

З допомогою компонентів JavaBean, що використовуються разом з сервлетами, можна виконати багато різних динамічним чином оброблюваних задач. Вони включають в себе організацію доступу до бази даних, використання корпоративних служб, таких як e-mail чи каталоги, а також управлінні потоками. Однак, одним з недоліків скриплетів є те, що вони ускладнюють підтримку сторінок JSP. З іншої сторони, технологія JSP забезпечує механізм, що має назву теги користувача, які дозволяють інкапсулювати динамічно виконувані функції і об'єкти, доступні через розширення мови JSP. Теги користувача упаковуються і розподіляються в одиниці, що називається бібліотекою тегів. Синтаксис тегів користувача не відрізняється від синтаксису, що використовується для елементів JSP, тобто <prefix:tag> є формою написання тегу користувача. На даний момент однією з найвідоміших і найпоширеніших бібліотек тегів користувача є JSTL.

4.2.4 Бібліотека JSTL

JSP Standard Tag Library (JSTL) – є стандартною бібліотекою тегів, що містить теги для управління поведінкою сторінки, теги форматування і інтернаціоналізації та допоміжні функції. JSTL є частиною Java EE API і більшості контейнерів servlet. Але для того, щоб використовувати JSTL в сторінках JSP, необхідно завантажити бібліотеки JSTL і додати їх в каталог WEB-INF/lib проекту.

Бібліотека JSTL містить п'ять різних видів тегів:

- core теги;
- теги форматування і локалізації;
- теги SQL;
- теги XML;
- функції JSTL.

Core теги, теги форматування і локалізації і функції JSTL є найбільш використовуваними серед всіх інших видів тегів.

Розглянемо як підключити на сторінку кожен з найбільш використовуваних видів тегів і більш детально, які є теги в кожному з видів.

Для того, щоб підключити кожен з вдів тегів необхідно скористатися директивою taglib. Приклади підключення core тегів, тегів форматування і локалізації і функцій JSTL представлений нижче:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

До core тегів відносяться теги представлені в таблиці 4.2.

Таблиця 4.2 – Core теги

Тег	Опис
1	2
<c:out >	Використовується для виведення даних на сторінку

Тег	Опис
1	2
<c:set >	Встановлює змінну на сторінці JSP
<c:remove >	Видаляє змінну
<c:catch>	Використовується для обробленні виключних ситуацій
<c:if>	Дозволяє організувати розгалуження на сторінці
<c:choose>	Аналог оператора switch в java. Використовується разом з тегами <c:when> і <c:otherwise>
<c:when>	Використовується разом з тегами <c:choose> і <c:otherwise> для реалізації оператору гілкування switch
<c:otherwise >	Використовується разом з тегами <c:choose> і <c:when> для реалізації оператору гілкування switch
<c:import>	Використовується для імпорту ресурсів в сторінку JSP
<c:forEach >	Використовується для організації циклів на сторінці. Підтримує можливість обходу колекцій
<c:forTokens>	Дає можливість обійти елементи рядка розділеного символом, вказаним в атрибуті тегу
<c:param>	Аналог тегу <jsp:patam>
<c:redirect >	Дає можливість передати управління іншій сторінці зі зміною початкової адреси запиту
<c:url>	Будує URL по ресурсам і параметрам

До тегів форматування і локалізації відносяться теги представлені в таблиці 4.3.

Таблиця 4.3 – Теги локалізації і форматування

Тег	Опис
1	2
<fmt:formatNumber>	Форматує число для виводу на екран
<fmt:parseNumber>	Перетворює рядок в число
<fmt:formatDate>	Форматує дату для виводу на екран
<fmt:parseDate>	Перетворює рядок в дату
<fmt:bundle>	Завантажує ресурси для використання в тілі тегу
<fmt:setLocale>	Встановлення значення локалі в змінній

Тег	Опис
1	2
<fmt:setBundle>	Завантажує ресурси і встановлює змінну для роботи з ними
<fmt:timeZone>	Повертає тайм зону.
<fmt:setTimeZone>	Встановлює тайм зону в змінну
<fmt:message>	Використовується для відображення інтернаціоналізованого повідомлення
<fmt:requestEncoding>	Встановлює кодування

До функцій JSTL відносяться функції представлені нижче:

fn:contains()
fn:containsIgnoreCase()
fn:endsWith()
fn:escapeXml()
fn:indexOf()
fn:join()
fn:length()
fn:replace()
fn:split()
fn:startsWith()
fn:substring()
fn:substringAfter()
fn:substringBefore()
fn:toLowerCase()
fn:toUpperCase()
fn:trim()

Всі функції призначені для роботи з рядками. Назва функції відповідає дії, яку вона виконує.

4.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення web-застосунку для роботи за базою даних, з допомогою якого можна переглядати записи в базі даних, додавати нові записи, видаляти і оновлювати існуючі записи. В якості прикладу, як і двох попередніх лабораторних роботах, був обраний об’єкт предметної області «інтернет-магазин» – продукт. Для створення web-інтерфейсу будуть використовуватися технології Servlet і JSP. В якості web-сервера буде використовуватися web-сервер Apache Tomcat 7.0.

1. Завантажте Apache Tomcat 7.0, або новішу версію з офіційного сайту у вигляді zip-архіву та розпакуйте його в будь-яке зручне місце.
2. Відкрийте IDE Eclipse та створіть новий Dynamic Web Project.
3. Перевірте наявність вкладки Servers у вікні IDE Eclipse. Якщо вкладка відсутня, натисніть в головному меню Window -> Show View -> Other, в діалоговому вікні в розділі Server оберіть Servers і натисніть кнопку Open.

Якщо ви раніше не налаштовували дану вкладку то вона буде виглядати як показано на рисунку 4.3 і необхідно виконати додаткові налаштування для конфігурації web-сервера.

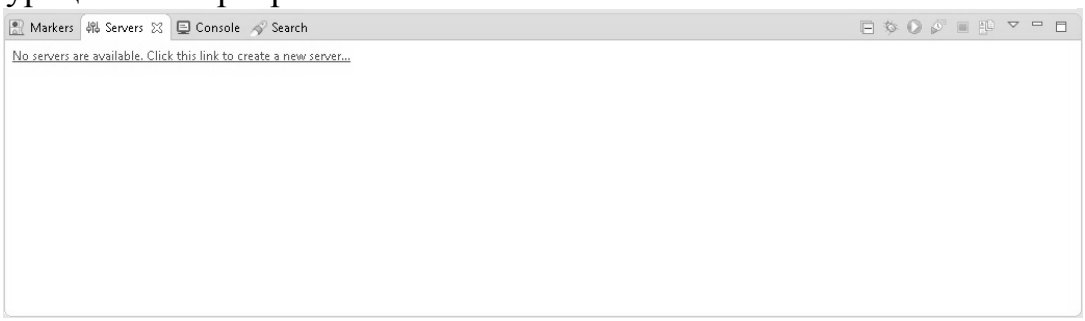

Рисунок 4.3 – Зовнішній вигляд вкладки Servers

4. У вкладці Servers натисніть на посилання No servers are available. Click this link to create a new server. В діалоговому вікні в розділі Apache оберіть Tomcat v7.0 Server та натисніть кнопку Next. В новому діалоговому вікні натисніть кнопку Browse та оберіть кореневий каталог завантаженого і розпакованого на кроці №1 серверу та натисніть кнопку Next. В наступному діалоговому вікні в полі Available оберіть проект, що був створений на кроці №2 і натисніть кнопку Add, а тоді кнопку Finish. Перед натисненням кнопки Finish діалогове вікно повинно мати вигляд, як показано на рисунку 4.4. Поле Available буде пустим, якщо у на момент конфігурації в каталозі з проектами був створений лише один Dynamic Web Project. Якщо таких проектів було створено декілька, то поле Available буде не пустим.

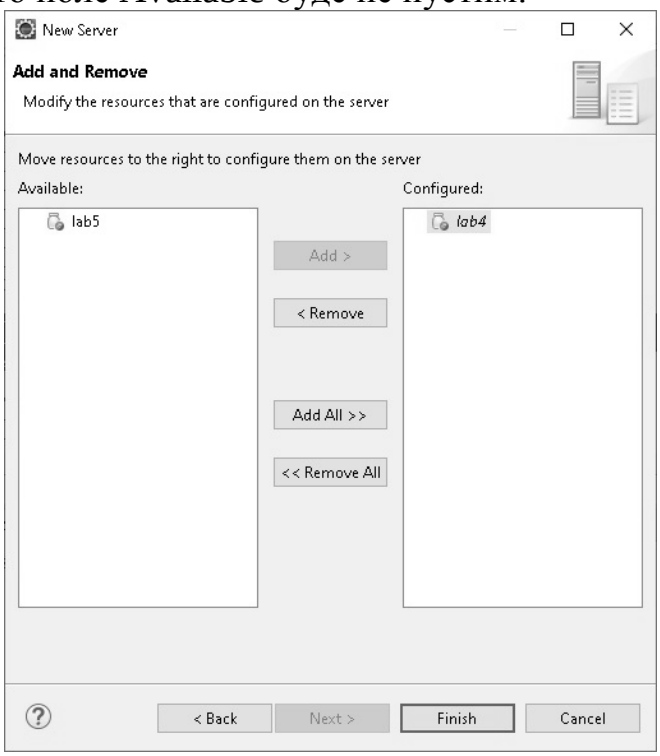

Рисунок 4.4 – Зовнішній вигляд діалогового вікна перед завершенням конфігурації

Після завершення налаштувань закладка Servers буде мати вигляд як показано на рисунку 4.5.

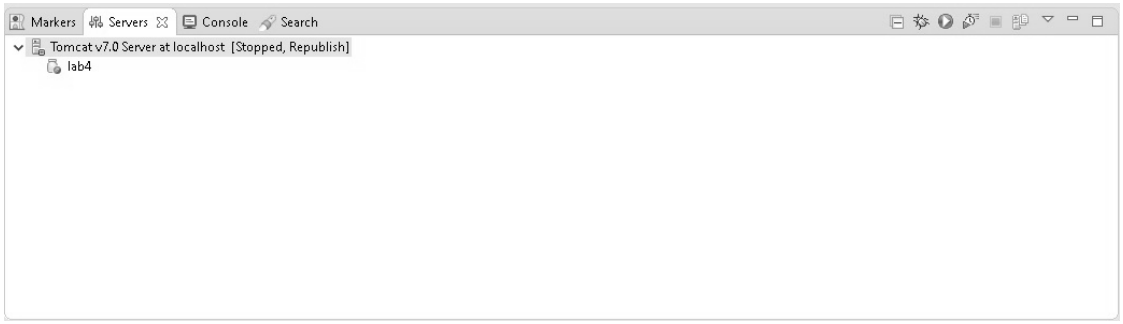

Рисунок 4.5 – Закладка Servers після завершення конфігурації

Після завершення конфігурації все ще можна додавати і видаляти проекти з використанням діалогового вікна з рисунку 4.2, доступ до якого можна отримати натиснувши правою кнопкою миші в закладці Servers на елемент Tomcat v7.0 at localhost, та обравши пункт меню Add and Remove. Можна додавати і завантажувати одразу декілька проектів з використанням одного сервера.

5. Далі необхідно перевірити чи конфігурація була виконана вірно. Створіть в каталозі WebContent новий JSP файл з іменем product.jsp і додайте в тег <body>, будь-який текст, наприклад «Hello from product page». Текст JSP файлу буде виглядати наступним чином:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  Hello from product page
</body>
</html>
```

В закладці Servers натисніть правою кнопкою миші на елемент Tomcat v7.0 Server at localhost та натисніть на елемент меню Debug. Дочекайтеся завершення старту сервера, відкрийте браузер та в рядку запиту введіть http://localhost:8080/lab4/product.jsp, де lab4 це контекст web-застосунку і він може відрізнятися в залежності від імені проекту. В браузері ви повинні побачити рядок «Hello from product page». Якщо сторінка завантажилася успішно, зупиніть сервер шляхом натиснення в контекстному меню Tomcat v7.0 at localhost елементу меню Stop.

6. В створеному проекті Dynamic Web Project відкрийте WebContent/WEB-INF та перевірте в даному каталозі наявність каталогу lib. Якщо, даного каталогу нема, то створіть його.

7. Перевірте наявність в каталозі WebContent/WEB-INF конфігураційного файлу web.xml. Якщо файл відсутній, натисніть правою кнопкою миші на проекті та в контекстному меню оберіть пункти Java EE Tools -> Generate Deployment Descriptor Stub. Відкрийте файл та перегляньте його

вміст. Даний конфігураційних файл мастить налаштування web-застосунку, які ми будемо змінювати в процесі створення нашого застосунку.

8. Додайте в корінь проекту файл build.gradle. Даний файл буде використовуватися для завантаження всіх необхідних для застосунку бібліотек з використанням системи автоматизованого збирання Gradle. Для виконання даного файлу необхідно попередньо встановити систему автоматизованого збирання Gradle, як показано в лабораторній роботі №2. Контент файлу build.gradle представлений нижче:

```
apply plugin: 'war'

repositories {
  mavenCentral()
}

sourceSets {
  main {
    java {
      srcDir 'src'
    }
    java.outputDir = file('WebContent/WEB-INF/classes')
  }
}

dependencies {
  runtime group: 'javax.servlet', name: 'javax.servlet-api', version: '3.0.1'
  runtime group: 'javax.servlet.jsp', name: 'jsp-api', version: '2.1'
  runtime group: 'javax.servlet', name: 'jstl', version: '1.2'
  runtime group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
  runtime group: 'org.hibernate', name: 'hibernate-core',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-entitymanager',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-osgi',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-envers',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-c3p0',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-proxool',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-infinispan',
    version: '4.3.6.Final'
  runtime group: 'org.hibernate', name: 'hibernate-ehcache',
    version: '4.3.6.Final'
  runtime group: 'antlr', name: 'antlr', version: '2.7.6'
  runtime group: 'commons-collections', name: 'commons-collections',
    version: '3.1'
  runtime group: 'dom4j', name: 'dom4j', version: '1.6.1'
  runtime group: 'javassist', name: 'javassist', version: '3.4.GA'
```

```
  runtime group: 'javax.transaction', name: 'jta', version: '1.1'
  runtime group: 'org.slf4j', name: 'slf4j-api', version: '1.5.6'
  runtime group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.5.6'
}

task copyToLib(type: Copy) {
  from configurations.runtime
  into "WebContent/WEB-INF/lib"
}

project.afterEvaluate {
  tasks.build.dependsOn(copyToLib)
}
```

9. Відкрийте консоль, перейдіть до кореня проекту і виконайте команду gradle build.

10. Поновіть каталог WebContent/WEB-INF/lib і переконайтеся, що після виконання команди gradle build в ньому з'явилися бібліотеки.

11. Підключіть всі бібліотеки з каталогу WebContent/WEB-INF/lib в CLASSPATH проекту. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Libraries натисніть кнопку Add JARs та додайте всі бібліотеки з каталогу WebContent/WEB-INF/lib. Натисніть кнопку Apply and close.

12. Задайте каталог, куди будуть компілюватися класи проекту. Він також, як і каталог lib, повинен знаходитися в каталозі WebContent/WEB-INF. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Source в полі вводу Default output folder вкажіть lab4/WebContent/WEB-INF/classes (lab4 – це ім'я проекту і, якщо у вас інше ім'я проекту, то цю частину необхідно замінити на ім'я вашого проекту). Натисніть кнопку Apply and close.

13. В каталозі src створіть пакет ua.cn.stu.dao та ua.cn.stu.domain. Перенесіть в пакет ua.cn.stu.dao класи HibernateDAOFactory і ProductDAO, а в пакет ua.cn.stu.domain клас Product, що були створені в попередній лабораторній роботі №3.

14. Додайте до класу HibernateDAOFactory метод closeSession, що буде використовуватися для закриття сесії Hibernate при зупинці сервера:

```
public void closeSession() {
  getSession().close();
}
```

15. Додайте до класу ProductDAO метод getProductById, що буде використовуватися для отримання продукту по ідентифікатору:

```
public Product getProductById(Long productId) {
  Product product = (Product) session.get(Product.class, productId);
  return product;
}
```

16. Додайте в каталог src пакет ua.cn.stu.listener та додайте в створений пакет клас StartStopListener. Застосунок взаємодіє за базою даних з використанням технології Hibernate. При ініціалізації фабрики Hibernate

викачається чимало часу, тому бажано проводити її ініціалізацію при старті застосунку. Це можна зробити з використанням механізму слухачів, яким и є клас StartStopListener. Код класу приведений нижче:

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import ua.cn.stu.dao.HibernateDAOFactory;

public class StartStopListener implements ServletContextListener {

  @Override
  public void contextInitialized(ServletContextEvent arg0) {
    HibernateDAOFactory.getInstance().getSession();
  }

  @Override
  public void contextDestroyed(ServletContextEvent arg0) {
    HibernateDAOFactory.getInstance().closeSession();
  }
}
```

17. Додайте налаштування для класу-слухача, створеного на попередньому кроці в конфігураційний файл web.xml. Конфігураційний тег, приведений нижче, повинен знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<listener>
  <listener-class>ua.cn.stu.listener.StartStopListener</listener-class>
</listener>
```

18. Додайте в каталозі src пакет ua.cn.stu.servlet, і в даному пакеті створіть клас сервлет, що буде оброблювати POST запити на додання, видалення оновлення продуктів нашого застосунку:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import ua.cn.stu.dao.HibernateDAOFactory;
import ua.cn.stu.domain.Product;

public class ProductServlet extends HttpServlet {

  private static final long serialVersionUID = 1L;

  @Override
  protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String actionType = req.getParameter("actiontype");
    //Обробка запиту на додання продукту
    if ("addproduct".equalsIgnoreCase(actionType)) {
      String productname = req.getParameter("productname");
```

```
      String productdescription = req.getParameter("productdescription");
      Product product = new Product();
      product.setName(productname);
      product.setDescription(productdescription);
      HibernateDAOFactory.getInstance().getProductDAO().createProduct(product);
    //Обробка запиту на видалення продукту
    } else if ("deleteproduct".equalsIgnoreCase(actionType)) {
      String productIdString = req.getParameter("productid");
      Long productId = Long.parseLong(productIdString);
      HibernateDAOFactory.getInstance().getProductDAO()
        .deleteProductById(productId);
    //Обробка запиту на оновлення продукту
    } else if ("updateproduct".equalsIgnoreCase(actionType)) {
      String productIdString = req.getParameter("productid");
      Long productId = Long.parseLong(productIdString);
      String productname = req.getParameter("productname");
      String productdescription = req.getParameter("productdescription");
      Product product = HibernateDAOFactory.getInstance().getProductDAO()
        .getProductById(productId);
      product.setName(productname);
      product.setDescription(productdescription);
      HibernateDAOFactory.getInstance().getProductDAO()
        .updateProduct(product);
    }
    //Після обробки запиту перейти на сторінку продукту
    resp.sendRedirect("product.jsp");
  }
}
```

19. Додайте в каталозі src пакет ua.cn.stu.databean і в даному пакеті створіть клас з ім’ям ProductListDatabean і кодом як показано нижче:

```
import java.util.List;

import ua.cn.stu.dao.HibernateDAOFactory;
import ua.cn.stu.domain.Product;

public class ProductListDatabean {

  private List<Product> productList;

  public ProductListDatabean() {
    productList = HibernateDAOFactory.getInstance().getProductDAO()
      .getAllProducts();
  }

  public List<Product> getProductList() {
    return productList;
  }

  public void setProductList(List<Product> productList) {
    this.productList = productList;
```

```
    }
}
```

20. Додайте в конфігураційний файл web.xml конфігурацію для нового сервлету. Конфігураційні елементи повинні знаходитись на тому ж рівні, що і елементи <welcome-file-list> і доданий на попередніх кроках елемент <listener>:

```
<servlet>
  <servlet-name>productservlet</servlet-name>
  <servlet-class>ua.cn.stu.servlet.ProductServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>productservlet</servlet-name>
  <url-pattern>/product</url-pattern>
</servlet-mapping>
```

21. Замініть контент JSP сторінки product.jsp наступним чином:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html>
<head>
  <meta charset="ISO-8859-1">
  <title>Product list</title>
</head>
<body>
  <jsp:useBean id="productListBean"
    class="ua.cn.stu.databean.ProductListDatabean"/>
  <h1>Product page</h1>
  <form method="post" action="product">
    <table style="width:100%" border="1">
      <tr>
        <td>Id</td>
        <td>Name</td>
        <td>Description</td>
        <td>Action</td>
      </tr>
      <c:forEach var="product" items="${productListBean.productList}">
        <tr>
          <td>
            <c:out value="${product.id}"/>
          </td>
          <td>
            <c:out value="${product.name}"/>
          </td>
          <td>
            <c:out value="${product.description}"/>
          </td>
          <td>
            <c:url var="deleteUrl" value="/deleteproduct.jsp">
              <c:param name="productid" value="${product.id}"/>
```

```
            <c:param name="productname" value="${product.name}"/>
            <c:param name="productdescription"
              value="${product.description}"/>
          </c:url>
          <c:url var="updateUrl" value="/updateproduct.jsp">
            <c:param name="productid" value="${product.id}"/>
            <c:param name="productname" value="${product.name}"/>
            <c:param name="productdescription"
              value="${product.description}"/>
          </c:url>
          <p><a href="${deleteUrl}">Delete product</a></p>
          <p><a href="${updateUrl}">Update product</a></p>
        </td>
      </tr>
    </c:forEach>
    <tr>
      <td>
      </td>
      <td>
        <input type="text" name="productname"/>
      </td>
      <td>
        <input type="text" name="productdescription"/>
      </td>
      <td>
        <input type="hidden" name="actiontype" value="addproduct"/>
        <input type="submit" name="addproduct" value="Add product">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

22. Додайте в каталог WebContent JSP файл іменем deleteproduct.jsp і контентом як показано нижче:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Delete product</title>
</head>
<body>
  <p>Below product will be deleted:</p>
  <p>Product id ${param.productid}</p>
  <p>Product name ${param.productname}</p>
  <p>Product description ${param.productdescription}</p>
  <p>Could you please confirm?</p>
```

```
  <c:url var="productUrl" value="/product.jsp"/>
  <form method="post" action="product">
    <input type="hidden" name="actiontype" value="deleteproduct"/>
    <input type="hidden" name="productid" value="${param.productid}">
    <input type="submit" name="deleteproduct" value="Delete product">
    <input type= "button" onclick="window.location.href='${productUrl}'"
      value="Cancel">
  </form>
</body>
</html>
```

23. Додайте в каталог WebContent JSP файл іменем updateproduct.jsp і контентом як показано нижче:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Update product</title>
</head>
<body>
  <p>Please update product fields and press
    Update product button or press Cancel button.</p>
  <c:url var="productUrl" value="/product.jsp"/>
  <form method="post" action="product">
    <input type="hidden" name="actiontype" value="updateproduct"/>
    <input type="hidden" name="productid" value="${param.productid}"/>
    <p>Product id: <c:out value="${param.productid}"/></p>
    <p>Product name:
    <input type="text" name="productname" value="${param.productname}"/></p>
    <p>Product description:
    <input type="text" name="productdescription"
      value="${param.productdescription}"/></p>
    <input type="submit" name="deleteproduct" value="Update product"/>
    <input type= "button" onclick="window.location.href='${productUrl}'"
      value="Cancel">
  </form>
</body>
</html>
```

24. Перевірте працездатність застосунку. Запустіть сервер, відкрийте браузер і введіть в рядок запиту запит http://localhost:8080/lab4/product.jsp. Головне вікно програми має виглядати як на малюнку 4.6. Спробуйте додати нові продукти, видалити продукт, оновити продукт.

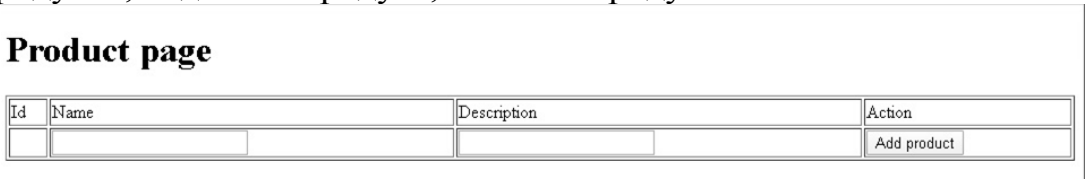

Рисунок 4.6 – Зовні вигляд web-застосунку в браузері

25. Вдоскональте застосунок відповідно до завдання на лабораторну роботу.

4.4 Завдання на лабораторну роботу

Вдосконалити застосунок, описаний в ході виконання лабораторної роботи, щоб він міг працювати з двома сутностями, на відміну від застосунку, описаного в лабораторній роботі. Предметну область необхідно вибрати з таблиці 4.4. Номер варіанта визначається по останній цифрі залікової книжки. Сутності повинні бути пов’язані відношенням один до багатьох.

Таблиця 4.4. Варіанти завдань

Номер варіанту	Предметна область
1	Піцерія (Офіціант, Відвідувач)
2	Крамниця (Постачальник, Товар)
3	Супермаркет (Продавець, Товар)
4	Відділення поліції (Поліцейський, Порушник)
5	Лікарня (Лікар, Пацієнт)
6	Кінотеатр (Кінозал, Відвідувач)
7	Корабель (Каюта, Пасажир)
8	Підприємство (Відділ, Співробітник)
9	Бібліотека (Книга, Читач)

4.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

4.6 Контрольні питання

1. Розкажіть про технологію Servlet;
2. Поясніть визначення Servlet-контейнер і JSP- контейнер;
3. Розкажіть що таке фільтри и їх можливості;
4. Розкажіть про технологію JSP;
5. Життєвий цикл JSP-сторінки;
6. Чим життєвий цикл JSP сторінки відрізняється від життєвого циклу сервлета;
7. Що таке скриплети і для чого вони використовуються;
8. Для чого використовується бібліотека JSTL;
9. Які види тегів є в бібліотеці JSTL;
10. Які базові теги є в бібліотеці JSTL.

4.7 Завдання до захисту лабораторної роботи

1. Реалізувати пошук доменних об’єктів зі сторони один по будь-якому полю з використанням запитів Native SQL.
2. Реалізувати пошук доменних об’єктів зі сторони один по будь-якому полю з використанням запитів за критерієм.
3. Реалізувати пошук доменних об’єктів зі сторони один по будь-якому полю з використанням запитів HQL.
4. Додати логіку по відображенню TOP 10 доменних об’єктів зі сторони багато з використання запитів Native SQL. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони один.
5. Додати логіку по відображенню TOP 10 доменних об’єктів зі сторони багато з використання запитів за критерієм. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони один.
6. Додати логіку по відображенню TOP 10 доменних об’єктів зі сторони багато з використання запитів HQL. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони один.
7. Додати серверну валідацію для полів вводу для доменного об’єкту зі сторони один. Виводити помилку валідації біля поля вводу, якщо воно містить помилку.
8. Створити тег користувача для виводу об’єктів зі сторони один.
9. Додати до web-застосунку локалізацію для декількох мов з використанням property файлів і тегів JSTL.
10. Замінити стандартні виклики сервлету для додання, видалення і заміни об’єктів на AJAX виклики (без перезавантаження сторінки) з використанням бібліотеки jQuery.

5 ЛАБОРАТОРНА РОБОТА №5
ВИВЧЕННЯ ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ REST-СЕРВІСІВ

5.1 Мета роботи

Вивчити технологій створення REST-сервісів, отримати практичні навички по створенню і застосуванню REST-сервісів.

5.2 Теоретичні відомості

5.2.1 Основи REST сервісів

REST визначає ряд архітектурних принципів проектування web-сервісів, орієнтованих на системні ресурси, включаючи способи обробки і передачі станів ресурсів по протоколу HTTP різноманітними застосунками, що написані на різних мовах програмування. На даний момент REST є однією з найпоширеніших моделей проектування web-сервісів і він, фактично, справив настільки великий вплив на Web, що практично витіснив дизайн інтерфейсів, заснований на SOAP і WSDL через більш простий стиль проектування.

Технологія REST не привернула великої уваги в 2000 році, коли Рой Філдінг (Roy Fielding) вперше представив її Каліфорнійському університеті в Ірвайні у своїй дисертації «Архітектурні стилі і дизайн мережевих архітектур програмного забезпечення», де аналізувався набір принципів архітектури програмного забезпечення, що використовував Web в якості платформи розподілених розрахунків. Однак сьогодні, через багато років, виникли і продовжують розвиватися велика кількість інфраструктур для REST. В чистому вигляді (в якому вона приваблює настільки щільну увагу) конкретна реалізація web-сервісів REST притримується чотирьох базових принципів проектування:

- явне використання методів протоколу HTTP;
- незбереження станів;
- ідентифікація ресурсів з використанням URI;
- передача даних в XML, JSON або в обох форматах даних.

Далі розглянемо кожен з принципів більш детально.

5.2.2 Зв'язок REST з протоколом HTTP

Однією з ключових характеристик web-сервісів RESTful є явне використання методів протоколу HTTP, що визначені в RFC 2616. Наприклад, HTTP GET визначається як метод генерації даних, що використовується клієнтським застосунком для отримання ресурсів та даних з web-сервера, або виконання запиту в надії на те, що web-сервер знайде і поверне набір відповідних ресурсів. REST пропонує розробникам використовувати HTTP-методи явно відповідно з визначенням протоколу. Цей основний принцип проектування REST встановлює однозначну відповідність між операціями create, read, update і delete (CRUD) і HTTP-методами. Згідно цієї відповідності для створення ресурсу на сервері використовується метод POST, для отримання ресурсу використовується метод GET, для зміни стану ресурсу чи для його

оновлення використовується метод PUT і для видалення ресурсу використовується метод DELETE.

Недоліком проектування багатьох Web API є використання HTTP-методів не за прямим призначенням. Наприклад, GET запит в HTTP зазвичай визначає один конкретний ресурс, або ж рядок запиту містить ряд параметрів, що визначають критерії пошуку сервером набору відповідних ресурсів. Саме так описаний метод GET в HTTP/1.1 RFC. Однак, часто зустрічаються неприваблівi Web API, що використовують HTTP GET для виконання різного роду транзакцій на сервері (наприклад для додання записів в базу даних). В таких випадках запит GET використовується некоректно, або, принаймні, не використовується в REST-стилі (RESTfully). Якщо Web API використовує GET для запуску віддалених процедур, запит може виглядати наступним чином:

```
GET /adduser?name=Robert HTTP/1.1
```

Це є невдалим дизайном, оскільки вищезгаданий метод підтримує операцію, що змінює стан, шляхом виконання HTTP-запиту GET. У випадку успішного виконання запиту в сховищі даних буде доданий новий користувач. Проблема тут є семантичною. Метод GET повинен використовуватися для повернення ресурсів, а не для додання запису в базу даних. З точки зору передбачуваного використання та з точки зору HTTP/1.1-сумісних web-сервісів таке використання методу GET є ненадійним. Але крім семантики ще однією проблемою є те, що метод GET використовується різними засобами web-кешування і пошуковими механізмами, які можуть виконувати ненавмисні зміни сервера шляхом простого обходу посилань, якщо метод GET використовувати для видалення, зміни і додання записів в базу даних або для зміни яких-небудь станів на стороні сервера.

Простим вирішенням даної проблем є передача імен і значень параметрів URI в тілі запиту і використання замість методу GET методу POST. Приклад запиту на додавання нового ресурсу після виконання вищезазначених змін буде виглядати наступним чином:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
 <name>Robert</name>
</user>
```

Це приклад RESTful-запиту: HTTP-запит POST використовується коректно, а тіло запиту містить корисне навантаження. Після цього клієнтський застосунок може отримати доданий ресурс, використовуючи новий URI, що вказує, що ресурс розміщений в каталозі /users:

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Це вірне використання запиту GET, оскільки він слугує тільки для отримання даних. GET – це операція, що повинна бути вільною від побічних

ефектів. Дана властивість ще відома під назвою ідемпотентність.

Аналогічний рефакторинг треба виконати в тих випадках, коли HTTP-запит GET підтримує операцію update:

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

Після рефакторингу даного запиту запит на зміну даних буде виглядати наступним чином:

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

Використання запиту PUT для заміни початкового ресурсу забезпечує значно більш прозорий інтерфейс, що сумісний з принципами REST і з визначенням HTTP-методів. Запит PUT з прикладу є явним в тому сенсі, що він вказує на ресурс, що оновлюється ресурс, визначаючи його в URI запиті і передає нове значення ресурсу від клієнта на сервер в тілі запиту PUT, замість того, щоб передавати атрибути ресурсу в вигляді слабо зв’язаного набору імен і значень параметрів в URI запиту.

Загальноприйнятим підходом, що відповідає рекомендаціям REST по явному застосуванню HTTP-методів є використання в URI іменників замість дієслів. В web-сервісі RESTful дієслова POST, GET, PUT і DELETE вже визначені протоколом. В ідеалі для реалізації узагальненого інтерфейсу і явного виклику операцій клієнтськими застосунками web-сервіс не повинен визначати додаткові команди чи віддалені процедури, наприклад /adduser чи /updateuser. Цей загальний принцип можна застосовувати і до тіла HTTP-запиту, яке призначене для передачі стану ресурсу, а не для імен методів, що викликаються.

5.2.3 Збереження стану REST сервісів

Для задоволення постійно зростаючих потреб до продуктивності web-сервіси REST повинні бути масштабованими. Для формування топології сервісів, яка дозволяє за необхідності перенаправляти запити з одного сервера на інший з метою зменшення загального часу реакції на виклик web-сервісу, зазвичай застосовують кластери серверів з можливістю розподілення навантаження і аварійного переключення на резерв, проксі-сервери і шлюзи. Використання проміжних серверів для покращення масштабованості потребує, щоб клієнти web-сервісів REST відправляли повні самодостатні запити, що містять всі необхідні для їх виконання дані., щоб компоненти а проміжних серверах могли пере направити, маршрутизувати і розподіляти навантаження без локального збереження стану між запитами.

При обробці повного самодостатнього запиту серверу не потрібно вивільняти стан чи контекст застосунку. Застосунок (чи клієнт) web-сервісу REST додає до HTTP-заголовку і тіло запиту всі параметри, контекст і дані, що

необхідні серверному компоненту для генерування відповіді. В цьому відношенні незбереження стану (statelessness) покращує продуктивність web-сервісу і спрощує дизайн і реалізацію серверних компонентів, оскільки відсутність стану на сервері усуває необхідність синхронізації сеансів даних з зовнішнім застосунком.

На рисунку 5.1 показаний сервіс, що зберігає стан, в якому застосунок може запросити наступну сторінку в багатосторінковому наборі результатів, вважаючи, що сервер зберігає послідовність переходів застосунку по цьому набору результатів. В цій моделі зі збереженням станів сервіс нарощує і зберігає змінну previousPage, щоб бути здатним відповідати на запити на отримання наступних сторінок.

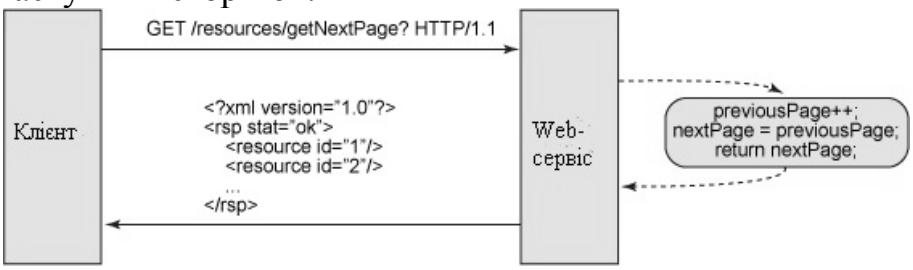

Рисунок 5.1 – Модель зі збереженням стану

Подібне сервіси, що зберігають стан, виходять складними. На платформі Java EE (Java Platform, Enterprise Edition) сервіси, що зберігаю стани, потребують великої кількості попередніх угод по ефективному збереженню і синхронізації сеансових даних в кластері Java EE-контейнерів. В середовищах такого типу розробники сервлетів або сторінок JSP і EJB-компонентів (Enterprise JavaBeans), стикаються з типовою проблемою пошуку причин виникнення виключної ситуації java.io.NotSerializableException при реалізації сеансів. Незалежно від джерела (контейнер сервлетів при реплікації HttpSession або EJB-контейнер при реплікації EJB-компоненту, що зберігає стан) така проблема може коштувати розробникам декількох днів пошуків в складному графі об’єктів, що визначають стан сервера, того єдиного об’єкту, в якому реалізується Serializable. Крім того, синхронізація сеансів збільшує накладні витрати, що негативно позначається на продуктивності сервера.

В протилежність цьому підходу, серверні компоненти, що не зберігають стан, більш прості в проектуванні, написанні і розподіленні між серверами зі збалансованим навантаженням. Сервіс, що не зберігає стан, не тільки краще працює, але і покладає основну відповідальність за збереження станів на клієнтський застосунок. В web-сервісі RESTful сервер відповідає за генерацію відповідей і за надання інтерфейсу, що дозволяє клієнтському застосунку самостійно зберігати свій стан. Наприклад, при запиті багатосторінкового набору результатів клієнтський застосунок повинен включити в запит номер конкретної сторінки, а не просто робити запит на наступну сторінку (див. рисунок 5.2).

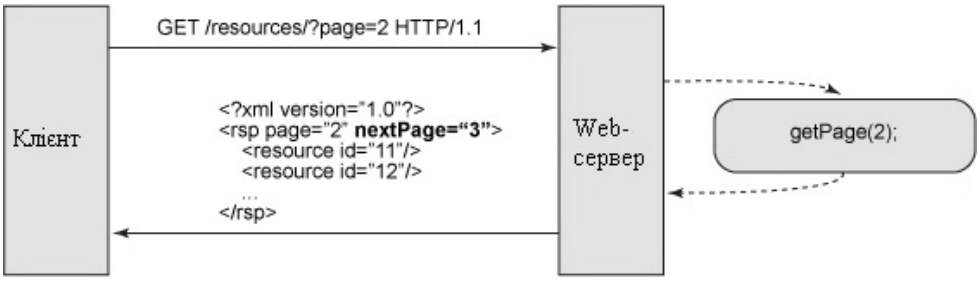

Рисунок 5.2 – Модель без збереження стану

Web-сервіс, що не зберігає стан, генерує відповідь, що містить посилання на номер наступної сторінки в наборі, що дозволяє клієнту самостійно подбати про збереження цього значення. В самому загальному вигляді цей аспект моделі web-сервісів RESTful можна розподілити на дві сфери відповідальності, що пояснюють суть функціонування сервісів, що не зберігають стан:

Сервер генерує відповіді, що містять посилання на інші ресурси для навігації застосунків по пов'язаним ресурсам. Такі відповіді містять вбудовані посилання. Аналогічним чином при запиті батьківського чи контейнерного ресурсу типова RESTful-відповідь може містити посилання на потомків батьківського елементу або на підлеглі ресурси., щоб зберігати зв'язок з ними.

Сервер генерує відповіді, що містять інформацію про те, чи підлягають вони кешуванню з метою підвищення продуктивності за рахунок зменшення кількості запитів ресурсів, що дублюються і повної відмови від деяких запитів. Для цього сервер додає до відповіді HTTP-заголовки Cache-Control і Last-Modified.

Клієнтський застосунок по заголовку Cache-Control відповіді визначає можливість кешування ресурсу. Також клієнтський застосунок читає заголовок Last-Modified відповіді і повертає значення дати в заголовку If-Modified-Since для відправки на сервер запиту про зміну ресурсу. Такий запит, так званий Conditional GET, використовує обидва заголовки. Якщо ресурс з вказаного часу не змінювався, відповіддю сервера є стандартний код 304 (Not Modified) і ресурс, що було запитано, не відправляється. Код відповіді 304 HTTP означає, що клієнтський застосунок може спокійно використовувати кешовану локальну копію ресурсу в якості самої останньої його версії, фактично опускаючи наступні запити GET до того часу, поки ресурс не буде змінений.

Клієнтський застосунок відправляє повні запити, які можуть оброблюватися незалежно від інших запитів. Це потребує від клієнтського застосунку використання в повному обсязі HTTP-заголовків, визначених інтерфейсом web-сервісу, і відправки повних представлень ресурсів в тілі запиту. Клієнтський застосунок відправляє запити, які практично нічого не знають про попередні запити, про існування сеансу на сервері, про здатність серверу додавати контекст в запит і про стан застосунку, що зберігається між запитами.

Така сумісна робота клієнтського застосунку і сервера дуже важлива для відмови від збереження стану в web-сервісах RESTful. Її результатом є рст продуктивності за рахунок зменшення трафіку і мінімізації стану серверного застосунку.

5.2.4 Ідентифікація ресурсів з використанням URI

З точки зору звернення до ресурсів з клієнтського застосунку надані URI визначають, наскільки інтуїтивним буде web-сервіс REST і чи буде він буде використовуватися так, як припускав розробник.

URI-адреса web-сервісу REST повинна бути інтуїтивно зрозумілою. URI треба розглядати як деякий самодокументований інтерфейс, який майже не потребує пояснень або звернення до розробника для його розуміння і для отримання відповідних ресурсів. Тому структура URI повинна бути простою, передбачуваною і зрозумілою.

Одним із способів досягнути такого рівня зручності використання – побудова URI по аналогії зі структурою каталогів. Такого роду URI є ієрархічними, такими, що виходять із одного кореневого шляху, розгалуження якого відображають основні функції сервісу. Відповідно до цього визначення, URI – це не просто рядок зі слешами в якості розділових знаків, а скоріше дерево з вище лежачими і нижче лежачими гілками, з'єднаними в вузлах. Наприклад, в сервісі обговорення різних тем, можна визначити структурований набір URI наступного вигляду:

```
http://www.myservice.org/discussion/topics/{topic}
```

Корінь /discussion має нижче стоячий вузол /topics. Нижче розташовуються назви тем (наприклад, gossip, technology і т. д.), кожна з яких вказує на свою гілку обговорення. В рамках даної структури можна легко викликати гілки обговорення простим введенням будь-чого після /topics/.

В деяких випадках каталогоподібна структура особливо добре підходить для шляхів до ресурсів. В якості прикладу можна назвати ресурси впорядковані по даті. Для них дуже добре підходить ієрархічний синтаксис. Наступний приклад інтуїтивно зрозумілий, оскільки заснований на правилах:

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

Перший фрагмент шляху – чотири цифри року, другий – дві цифри дня і третій – дві цифри місяця. Подібне пояснення може показатися дещо спрощеним, але це саме той рівень простоти, який нам потрібен. Люди і комп'ютери можуть легко генерувати подібні структуровані URI, оскільки вони засновані на правилах. Вказування фрагментів шляху у відповідних позиціях відповідно синтаксису робить URI уніфікованими, оскільки існує закономірність їх створення:

```
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}
```

Приведемо деякі додаткові рекомендації, на які слід звернути увагу при обдумуванні структури URI для web-сервісів RESTful:

1. Необхідно приховувати розширення файлів серверних сценаріїв (.jsp, .php, .asp), якщо такі використовуються, що можна було виконувати перенесення застосунків на інші технології без зміни URI.
2. Необхідно використовувати тільки малі літери.
3. Необхідно заміняти пробіли дефісами чи знаками підкреслення (чимось одним).
4. Треба максимально уникати використання рядків запиту.

5. Замість використання коду 404 Not Found для URI, що вказує на неповний шлях, завжди необхідно надавати в якості відповіді ресурс чи сторінку за замовчуванням.

6. URI також повинен бути статичним, щоб при зміні ресурсів чи реалізації сервісу, посилання залишалося тим самим. Це дозволяє зберігати закладки. Також важливо, щоб взаємозв'язки між ресурсами, закодованими в URI, залишалися незалежними від способу вказування місця розташування ресурсів в сховищі.

5.2.5 Формати передачі даних

Представлення ресурсу зазвичай відображає поточний стан ресурсу (або його атрибутів) на момент запиту його клієнтським застосунком. Представлення ресурсів в цьому сенсі є просто знімками стані в певний момент часу. Ці представлення повинні бути такими ж простими, як представлення запису базі даних, що складається з відображення між іменем колонок і XML-тегами, де значення елементів в XML містить значення рядків. Якщо система має модель даних, то відповідно цьому визначенню представлення ресурсу є знімком стану атрибутів одного з об'єктів моделі даних системи. Це ті об'єкти, які буде обслуговувати web-сервіс REST.

Останній набір обмежень, тісно пов'язаний з дизайном web-сервісів RESTful, відноситься до формату даних, якими обмінюються застосунок і сервіс при роботі в режимі запит-відповідь. Тут особливо важлива простота, читабельність і зв'язаність. Об'єкти моделі зазвичай якимось чином пов'язані і ці відношення між об'єктами (ресурсами) моделі даних повинні відображатися в способі їх представлення для передачі клієнтському застосунку. В сервісі обговорень приклад представлень зв'язаних ресурсів може включати в себе кореневу тему обговорення і її атрибути, а також вбудовані посилання на відповіді, відправлені в цю тему.

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

Також, щоб надати клієнтським застосункам можливість робити запит на конкретний найбільш підходящий їх тип вмісту, необхідно проектувати сервіс так, що він використовував вбудований HTTP-заголовок Accept, значення якого є MIME типом. Деякими загальновживаними MIME-типами, що використовуються в RESTful-сервісах є JSON (application/json), XML (application/xml), XHTML (application/xhtml+xml) та інші.

Це дозволить використовувати сервіс клієнтським застосункам, написаним на різних мовах, працюючих на різних платформах і приладах.

Використання MIME-типів і HTTP-заголовка Accept являє собою

механізм узгодження вмісту (content negotiation), що дозволяє клієнтським застосункам обирати підходящий для них формат даних і мінімізувати зв’язність даних між сервісом і застосунком, що його використовує.

5.2.6 HTTP коди

В стандарті HTTP описано більш ніж 70 статус кодів. Гарним підходом є використання хоча б основних статус кодів.

200 – ОК – успішний запит. Якщо клієнт зробив запит на деякі дані, то вони знаходяться в заголовку і/або тілі повідомлення.

201 – ОК – в результаті успішного виконання запиту був створений новий ресурс.

204 – ОК – ресурс успішно видалений.

304 – Not Modified – клієнт може використовувати дані з кеша.

400 – Bad Request – запит інвалідний або не може бути оброблений.

401 – Unauthorized – запит потребує аутентифікацію користувача.

403 – Forbidden – сервер зрозумів запит, але відмовляється його обробити або доступ заборонений.

404 – Not found – ресурс не знайдений.

500 – Internal Server Error – розробники API повинні намагатися уникати таких помилок. Ці помилки повинні бути виловлені в глобальному catch блоці, залогова ні, але не повинні бути повернені у відповіді.

Чим великим є набір кодів, який буде використовуватися, тим більш зрозумілим буде API, яке створюється. Але треба врахувати, що деякі коди браузери оброблюють по-різному. Наприклад, деякі браузери отримавши код відповіді 307

Одразу виконують перенаправлення, а деякі дозволяють обробити таку ситуацію і відмінити дію. Перш ніж використовувати той чи інший код, необхідно цілком розуміти, як він буде оброблюватися клієнтською стороною.

5.2.7 Параметри пошуку ресурсів

Щоб спростити використання сервісів, що відповідають за повернення будь-яких даних і додатково зробити їх найбільш продуктивними, необхідно використовувати в якості параметрів запиту параметри для сортування, фільтрації, вибору полів і пагінації.

Для виконання фільтрації необхідно використовувати унікальний параметр запиту для кожного поля. Це дозволяє обмежити кількість даних, що виводиться, що оптимізує час обробки запиту. Наприклад, щоб вивести всі червоні книжки, необхідно виконати запит:

```
GET /books?color=red
```

Сортування реалізується подібно фільтрації. Наприклад, щоб вивести всі книжки, відсортовані по року публікації по спаданню і по назві по зростанню необхідно виконати наступний запит:

```
GET /books?sort=-year,+name
```

Для того, щоб підтримати можливість завантаження списку ресурсів, які

повинні відображатися на певній сторінці застосунку, в REST API повинен бути передбачений функціонал пагінації. Реалізується він з допомогою знайомих нам по SQL параметрів limit і offset. Наприклад:

```
GET /books?offset=10&limit=5
```

Окрім того гарним тоном є вивід посилання на попередню і наступну , першу і останню сторінки в хедері Link. Наприклад:

```
Link: <http://localhost/api/books?offset=15&limit=5>; rel="next",
<http://localhost/api/books?offset=50&limit=3>; rel="last",
<http://localhost/api/books?offset=0&limit=5>; rel="first",
<http://localhost/api/books?offset=5&limit=5>; rel="prev"
```

Рекомендовано також повертати загальну кількість ресурсів в хедері X-Total-Count.

Для більш зручного використання сервісу, для економії трафіку можна надати можливість управляти форматом виводу даних, що реалізується наданням можливості вибору полів ресурсу, які повинен повернути REST сервіс. Наприклад, якщо необхідно отримати тільки id книжок і їх кольори, необхідно виконати наступний запит:

```
GET /books?fields=id,color
```

5.2.8 Обробка виключень

При виникненні помилкових ситуацій необхідно виводити відформатовану і зрозумілу інформацію. Це відноситься в першу чергу до статус кодів в HTTP відповіді. Помилки в сервісах найчастіше відносяться до двох типів:

- 4xx – помилки клієнта;
- 5xx – помилки сервера.

У випадку помилки клієнта, наприклад, помилка валідації якого-небудь з параметрів запиту, в тілі відповіді рекомендовано передавати корисну інформацію про помилку: повідомлення, опис, код (наприклад в форматі JSON). У випадку помилки сервера відправляти додаткову інформацію в тілі відповідне завжди можливо, наприклад у випадках коли сервер не доступний.

Поганим тоном є вивід всього стек трейсу виключення. Рекомендовано для кожної виключної ситуації мати свій код. В подальшому для виводу інформації про помилку можна буде додавати посилання на документацію, в якій даний код буде виступати унікальним ідентифікатором. Наприклад:

```
{
   "code" : 1234,
   "message" : "Something bad happened :(",
   "description" : "More details about the error here",
   "moreInfo": "http:/localhost/api/v2/errors/1234"
}
```

5.2.9 Анотації JAX-RS

До анотацій JAX-RS відносяться анотації @Path, @GET, @POST, @PUT, @DELETE, @PathParam, @QueryParam, @FormParam, @Consumes, @Produces.

Розглянемо призначення і приклад використання кожної з анотацій.

Анотація @GET використовується для маркування методу, що буде відгукуватися на GET запити від клієнта:

```
@GET
public String getHTML() {
  ...
}
```

Анотація @Produces визначає тип відгуку, що повертає метод, промаркований даною анотацією:

```
@GET
@Produces("application/xml")
public Contact getXML() {
  ...
}

@GET
@Produces("application/json")
public Contact getJSON() {
  ...
}
```

Анотація @Path визначає URI, на який буде відгукуватися метод, над яким вказана дана анотація:

```
@GET
@Produces("application/xml")
@Path("xml/product")
public Contact getProductXML() {
  ...
}
```

Анотація @PathParam використовується для зв’язування параметрів, що передаються через URI (REST стиль передачі параметрів) з параметрами методу:

```
@GET
@Produces("application/xml")
@Path("xml/{firstName}")
public Contact getXML(@PathParam("firstName") String firstName) {
  Contact contact = contactService.findByFirstName(firstName);
  return contact;
}

@GET
@Produces("application/json")
@Path("json/{firstName}")
public Contact getJSON(@PathParam("firstName") String firstName) {
  Contact contact = contactService.findByFirstName(firstName);
  return contact;
}
```

Анотація @QueryParam використовується для зв’язування параметрів GET запиту з параметрами методу:

```
@GET
@Produces("application/json")
@Path("json/companyList")
public CompanyList getJSON(@QueryParam("start") int start,
  @QueryParam("limit") int limit) {
  CompanyList list = new CompanyList(companyService.listCompanies(start,
    limit));
  return list;
}
```

Анотація @POST використовується для маркування методу, що буде відгукуватися на POST запити від клієнта:

```
@POST
@Consumes("application/json")
@Produces("application/json")
public RestResponse<Contact> create(Contact contact) {
  ...
}
```

Анотація @Consumes використовується для встановлення MIME медіа типів, які може приймати на вхід REST сервіс:

```
@PUT
@Consumes("application/json")
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> update(Contact contact) {
  ...
}
```

Анотація @FormParam використовується для зв’язування параметрів POST запиту з параметрами методу (аналог @QueryParam тільки для POST запитів):

```
@POST
public String save(@FormParam("firstName") String firstName,
  @FormParam("lastName") String lastName) {
  ...
}
```

Анотація @PUT – використовується для маркування методу, що буде відгукуватися на PUT запити від клієнта:

```
@PUT
@Consumes("application/json")
@Produces("application/json")
@Path("{contactId}")
public RestResponse<Contact> update(Contact contact) {
  ...
}
```

Анотація @DELETE - використовується для маркування методу, що буде відгукуватися на DELETE запити від клієнта:

```
@DELETE
@Produces("application/json")
@Path("{contactId}")
```

```
public RestResponse<Contact> delete(@PathParam("contactId") int contactId) {
  ...
}
```

5.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення web-застосунку для роботи за базою даних, з допомогою якого можна переглядати записи в базі даних, додавати нові записи, видаляти і оновлювати існуючі записи. В якості прикладу, як і в попередніх лабораторних роботах, був обраний об’єкт предметної області «інтернет-магазин» – продукт. Для створення web-інтерфейсу використовується технологія JSP, а для обробки запитів користувача використовуються REST-сервіси, створені з використанням бібліотеки Jersey. В якості web-сервера використовується web-сервер Apache Tomcat 7.0. На відміну від попередньої лабораторної роботи, всі запити до сервера здійснюються з використанням AJAX. Для здійснення AJAX запитів використовується бібліотека jQuery.

1. Відкрийте IDE Eclipse і створіть новий Dynamic Web Project.
2. Виконайте конфігурацію вкладки Servers, якщо це не було зроблено в попередній лабораторній роботі (див. кроки 1, 3 та 4 в порядку виконання лабораторної роботи №4).
3. Виконайте конфігурацію проекту для завантаження його на сервері Apache Tomcat. У вкладці Servers натисніть правою кнопкою миші на елементі дерева Tomcat v7.0 Server at localhost і в контекстному меню натисніть на елемент Add and Remove. В діалоговому вікні перемістіть створений проект з поля Available в поле Configured, а всі інші проекти, що знаходяться в полі Configured перемістіть в поле Available , як показано на рисунку 5.3. Натисніть кнопку Finish.

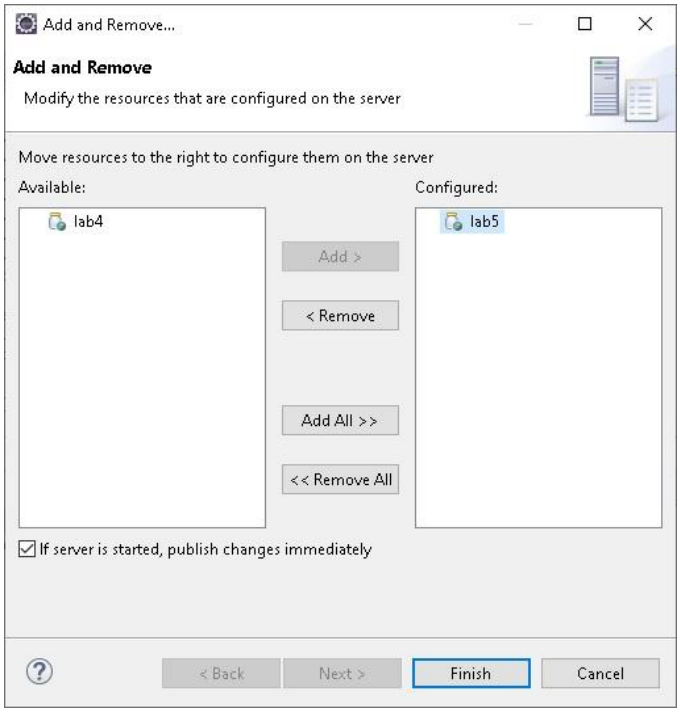

Рисунок 5.3 – Конфігурація проекту для завантаження на сервері

4. Далі необхідно перевірити чи конфігурація була виконана вірно. Створіть в каталозі WebContent новий JSP файл з іменем product.jsp і додайте в тег <body>, будь-який текст, наприклад «Hello from product page». Текст JSP файлу буде виглядати наступним чином:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  Hello from product page
</body>
</html>
```

5. В закладці Servers натисніть правою кнопкою миші на елемент Tomcat v7.0 Server at localhost та натисніть на елемент меню Debug. Дочекайтеся завершення старту сервера, відкрийте браузер та в рядку запиту введіть http://localhost:8080/lab5/product.jsp, де lab5 це контекст web-застосунку і він може відрізнятися в залежності від імені проекту. В браузері ви повинні побачити рядок «Hello from product page». Якщо сторінка завантажилася успішно, зупиніть сервер шляхом натиснення в контекстному меню Tomcat v7.0 at localhost елементу меню Stop.

6. В створеному проекті Dynamic Web Project відкрийте WebContent/WEB-INF та перевірте в даному каталозі наявність каталогу lib. Якщо, даного каталогу нема, то створіть його.

7. Перевірте наявність в каталозі WebContent/WEB-INF конфігураційного файлу web.xml. Якщо файл відсутній, натисніть правою кнопкою миші на проекті та в контекстному меню оберіть пункти Java EE Tools -> Generate Deployment Descriptor Stub. Відкрийте файл та перегляньте його вміст. Даний конфігураційних файл містить налаштування web-застосунку, які ми будемо змінювати в процесі створення нашого застосунку.

8. Додайте в корінь проекту файл build.gradle. Даний файл буде використовуватися для завантаження всіх необхідних для застосунку бібліотек з використанням системи автоматизованого збирання Gradle. Для виконання даного файлу необхідно попередньо встановити систему автоматизованого збирання Gradle, як показано в лабораторній роботі №2. Контент файлу build.gradle представлений нижче:

```
apply plugin: 'war'

repositories {
  mavenCentral()
}

sourceSets {
  main {
```

```
    java {
      srcDir 'src'
    }
    java.outputDir = file('WebContent/WEB-INF/classes')
  }
}

dependencies {
  compile group: 'javax.servlet', name: 'javax.servlet-api', version: '3.0.1'
  compile group: 'javax.servlet.jsp', name: 'jsp-api', version: '2.1'
  compile group: 'javax.servlet', name: 'jstl', version: '1.2'
  compile group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
  compile group: 'org.hibernate', name: 'hibernate-core',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-entitymanager',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-osgi',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-envers',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-c3p0',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-proxool',
    version: '4.3.6.Final'
        compile group: 'org.hibernate', name: 'hibernate-infinispan', version:
'4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-ehcache',
    version: '4.3.6.Final'
  compile group: 'antlr', name: 'antlr', version: '2.7.6'
  compile group: 'commons-collections', name: 'commons-collections',
    version: '3.1'
  compile group: 'dom4j', name: 'dom4j', version: '1.6.1'
  compile group: 'javassist', name: 'javassist', version: '3.4.GA'
  compile group: 'javax.transaction', name: 'jta', version: '1.1'
  compile group: 'org.slf4j', name: 'slf4j-api', version: '1.5.6'
  compile group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.5.6'
  compile group: 'org.glassfish.jersey.containers',
    name:'jersey-container-servlet', version :'2.30.1'
  compile group: 'org.glassfish.jersey.media',
    name: 'jersey-media-json-jackson', version: '2.30.1'
  compile group: 'org.glassfish.jersey.inject', name: 'jersey-hk2',
    version: '2.30.1'
  compile group: 'org.glassfish.jersey.media',
    name: 'jersey-media-multipart', version: '2.30.1'
  compile group: 'io.swagger', name: 'swagger-jaxrs', version: '1.5.13'
}

task copyToLib(type: Copy) {
  from configurations.runtime
  into "WebContent/WEB-INF/lib"
}
```

```
project.afterEvaluate {
  tasks.build.dependsOn(copyToLib)
}
```

9. Відкрийте консоль, перейдіть до кореня проекту і виконайте команду gradle build.

10. Поновіть каталог WebContent/WEB-INF/lib і переконайтеся, що після виконання команди gradle build в ньому з'явилися бібліотеки.

11. Підключіть всі бібліотеки з каталогу WebContent/WEB-INF/lib в CLASSPATH проекту. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Libraries натисніть кнопку Add JARs та додайте всі бібліотеки з каталогу WebContent/WEB-INF/lib. Натисніть кнопку Apply and close.

12. Задайте каталог, куди будуть компілюватися класи проекту. Він також, як і каталог lib, повинен знаходитися в каталозі WebContent/WEB-INF. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Source в полі вводу Default output folder вкажіть lab5/WebContent/WEB-INF/classes (lab5 – це ім'я проекту і, якщо у вас інше ім'я проекту, то цю частину необхідно замінити на ім'я вашого проекту). Натисніть кнопку Apply and close.

13. В каталозі src створіть пакети ua.cn.stu.dao та ua.cn.stu.domain. Перенесіть в пакет ua.cn.stu.dao класи HibernateDAOFactory і ProductDAO, а в пакет ua.cn.stu.domain клас Product, що були створені в попередній лабораторній роботі №4.

14. В каталозі src створіть пакет ua.cn.stu.listener. Перенесіть в нього клас StartStopListener, створений в попередній лабораторній роботі.

15. Додайте налаштування для класу-слухача, створеного на попередньому кроці в конфігураційний файл web.xml. Конфігураційний тег, приведений нижче, повинен знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<listener>
  <listener-class>ua.cn.stu.listener.StartStopListener</listener-class>
</listener>
```

16. Додайте product.jsp сторінку в список файлів за замовчуванням для того, щоб ця сторінка була доступна з кореня застосунку без вказування її імені. Для цього в файлі web.xml додайте в елемент welcome-file-list наступний елемент:

```
<welcome-file>product.jsp</welcome-file>
```

17. В каталозі src створіть пакет ua.cn.stu.service і створіть в ньому сервісний клас ProductService:

```
import java.util.List;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
```

```
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import ua.cn.stu.dao.HibernateDAOFactory;
import ua.cn.stu.domain.Product;

@Path("product")
public class ProductService {

  @GET
  @Path("getAllProducts")
  @Produces(MediaType.APPLICATION_JSON)
  public List<Product> getAllProducts() {
    return HibernateDAOFactory.getInstance().getProductDAO().getAllProducts();
  }

  @PUT
  @Path("addProduct/name/{productname}/description/{productdescription}")
  public void addProduct(@PathParam("productname") String productname,
    @PathParam("productdescription") String productdescription) {
    Product product = new Product();
    product.setName(productname);
    product.setDescription(productdescription);
    HibernateDAOFactory.getInstance().getProductDAO().createProduct(product);
  }

  @DELETE
  @Path("deleteProduct/{productid}")
  public void deleteProduct(@PathParam("productid") String productid) {
    Long productId = Long.parseLong(productid);
    HibernateDAOFactory.getInstance().getProductDAO()
      .deleteProductById(productId);
  }

  @POST

@Path("updateProduct/id/{productid}/name/{productname}/description/{productdesc
ription}")
  public void updateProduct(@PathParam("productid") String productid,
    @PathParam("productname") String productname,
    @PathParam("productdescription") String productdescription) {
    Long productId = Long.parseLong(productid);
    Product product = HibernateDAOFactory.getInstance().getProductDAO()
      .getProductById(productId);
    product.setName(productname);
    product.setDescription(productdescription);
    HibernateDAOFactory.getInstance().getProductDAO().updateProduct(product);
  }
}
```

18. Додайте конфігурацію для сервлету Jersey в конфігураційний файл web.xml. Конфігураційні теги, приведені нижче, повинні знаходитись всередині

тегу <web-app> на одному рівні з тегом <welcome-file-list>. Зверніть увагу, що значення конфігураційного параметру jersey.config.server.provider.packages повинно відповідати імені пакету, де знаходиться клас ProductService:

```
<servlet>
  <servlet-name>jersey-serlvet</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>ua.cn.stu.service</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
  <servlet-name>jersey-serlvet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
```

19. Створіть в каталозі WebContent каталог з іменем js.

20. Завантажте з офіційного сайту jQuery jquery.com Java Script бібліотеку jQuery і розмістіть її в каталозі js.

21. Замініть контент JSP сторінки product.jsp на контент, представлений нижче:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Product page</title>
<script src="js/jquery-3.5.0.js"></script>
<script type="text/javascript">
  jQuery( document ).ready(function() {
    //Send ajax request for show product list
    jQuery.ajax({
      url: 'rest/product/getAllProducts',
      dataType: 'json',
      type: 'GET',
      success: function(data) {
        jQuery('#headerrow').after(createDataRowsFromJson(data));
      }
    });
    jQuery( "#addproduct").click(function() {
      var url = 'rest/product/addProduct/name/' +
        jQuery("#productname").val() + '/description/' +
        jQuery("#productdescription").val();
      //Send ajax request for adding new product
      jQuery.ajax({
        url: url,
        dataType: 'json',
        type: 'PUT',
```

```
      success: function(data) {
        //Send ajax request for refresh product list after adding product
        jQuery.ajax({
          url: 'rest/product/getAllProducts',
          dataType: 'json',
          type: 'GET',
          success: function(data) {
            jQuery('.datarow').remove();
            jQuery('#headerrow').after(createDataRowsFromJson(data));
            jQuery("#productname").val('');
            jQuery("#productdescription").val('');
          }
        });
      }
    });
  });
});

function createDataRowsFromJson(data) {
  var tableContent = "";
  for (var key in data) {
    if (data.hasOwnProperty(key)) {
      tableContent = tableContent + "<tr class='datarow'>";
      tableContent = tableContent + "<td>";
      tableContent = tableContent + data[key].id;
      tableContent = tableContent + "</td>";
      tableContent = tableContent + "<td>";
      tableContent = tableContent + data[key].name;
      tableContent = tableContent + "</td>";
      tableContent = tableContent + "<td>";
      tableContent = tableContent + data[key].description;
      tableContent = tableContent + "</td>";
      tableContent = tableContent + "<td>";
      tableContent = tableContent + "<input type='button'
        onclick='deleteProduct(" + data[key].id + ")'
        value='Delete product'/>";
      tableContent = tableContent + "<input type='button'
        onclick='editProduct(this)' value='Edit product'/>";
      tableContent = tableContent + "</td>";
      tableContent = tableContent + "</tr>";
    }
  }
  return tableContent;
}

function deleteProduct(id) {
  var url = 'rest/product/deleteProduct/' + id;
  //Send ajax request for deleting product
  jQuery.ajax({
    url: url,
```

```
    dataType: 'json',
    type: 'DELETE',
    success: function(data) {
      jQuery.ajax({
        url: 'rest/product/getAllProducts',
        dataType: 'json',
        type: 'GET',
        success: function(data) {
          jQuery('.datarow').remove();
          jQuery('#headerrow').after(createDataRowsFromJson(data));
        }
      });
    }
  });
}

function editProduct(button) {
  jQuery(button).closest('tr').children().each(function(index, value){
    if(index == 1){
      jQuery(this).html("<input type='text' id='editname' value='" +
        jQuery(this).text() + "'/>");
    } else if(index == 2){
      jQuery(this).html("<input type='text' id='editdescription' value='" +
        jQuery(this).text() + "'/>");
    } else if (index == 3){
      var actionHtml = "<input type='button' onclick='applyEditProduct(this)'
        value='Update product'/>"
      actionHtml = actionHtml + "<input type='button'
        onclick='cancelEdit(this)' value='Cancel edit'/>"
      jQuery(this).html(actionHtml);
    }
  });
}

function cancelEdit(button) {
  var id;
  jQuery(button).closest('tr').children().each(function(index, value){
    if(index == 0){
      id = jQuery(this).text();
    } else if(index != 0 && index != 3){
      jQuery(this).html(jQuery(this).find('input').val());
    } else if (index == 3){
      var actionHtml = "<input type='button' onclick='deleteProduct(" + id +
        ")' value='Delete product'/>"
      actionHtml = actionHtml + "<input type='button'
        onclick='editProduct(this)' value='Edit product'/>"
      jQuery(this).html(actionHtml);
    }
  });
}
```

```
  function applyEditProduct(button){
    var id, name, description;
    jQuery(button).closest('tr').children().each(function(index, value){
      if(index == 0){
        id = jQuery(this).text();
      } else if(index == 1){
        name = jQuery(this).find('input').val();
      } else if (index == 2){
        description = jQuery(this).find('input').val();
      }
    });
    //Send ajax request for upating product
    var url = 'rest/product/updateProduct/id/' + id + '/name/' + name +
      '/description/' + description;
    jQuery.ajax({
      url: url,
      dataType: 'json',
      type: 'POST',
      success: function(data) {
        jQuery.ajax({
          url: 'rest/product/getAllProducts',
          dataType: 'json',
          type: 'GET',
          success: function(data) {
            jQuery('.datarow').remove();
            jQuery('#headerrow').after(createDataRowsFromJson(data));
          }
        });
      }
    });
  }
</script>
</head>
<body>
  <h1>Product page</h1>
  <table style="width: 100%" border="1" id="producttable">
    <tr id="headerrow">
      <td>Id</td>
      <td>Name</td>
      <td>Description</td>
      <td>Action</td>
    </tr>
    <tr>
      <td></td>
      <td><input type="text" name="productname" id="productname" /></td>
      <td><input type="text" name="productdescription"
        id="productdescription" /></td>
      <td><input type="button" name="addproduct" id="addproduct"
        value="Add product"></td>
```

```
    </tr>
  </table>
</body>
</html>
```

22. Перевірте працездатність застосунку. Запустіть сервер, відкрийте браузер і введіть в рядок запиту запит http://localhost:8080/lab5. Головне вікно застосунку має виглядати як на малюнку 5.4. Спробуйте додати новий продукт, видалити продукт, змінити продукт.

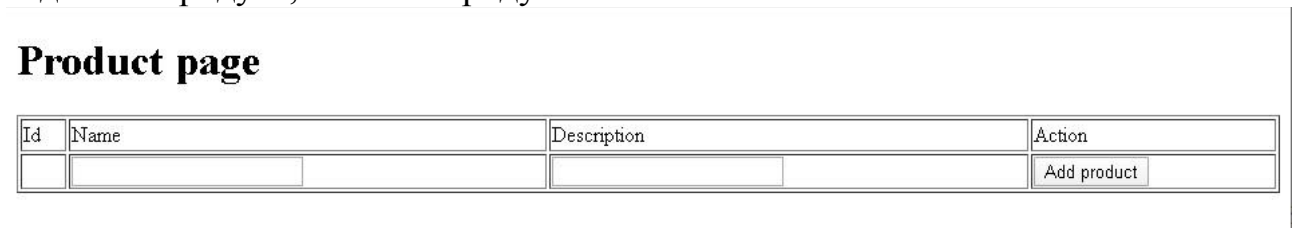

Рисунок 5.4 – Головне вікно застосунку

23. Вдоскональте застосунок відповідно до завдання на лабораторну роботу.

5.4 Завдання на лабораторну роботу

Вдосконалити застосунок, описаний в ході виконання лабораторної роботи, щоб він міг працювати з двома сутностями, на відміну від застосунку, описаного в лабораторній роботі. Предметну область необхідно вибрати з таблиці 5.1. Номер варіанта визначається по останній цифрі залікової книжки. Сутності повинні бути пов’язані відношенням один до багатьох.

Таблиця 5.1. Варіанти завдань

Номер варіанту	Предметна область
1	Піцерія (Офіціант, Відвідувач)
2	Крамниця (Постачальник, Товар)
3	Супермаркет (Продавець, Товар)
4	Відділення поліції (Поліцейський, Порушник)
5	Лікарня (Лікар, Пацієнт)
6	Кінотеатр (Кінозал, Відвідувач)
7	Корабель (Каюта, Пасажир)
8	Підприємство (Відділ, Співробітник)
9	Бібліотека (Книга, Читач)

5.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

5.6 Контрольні питання

1. Що таке REST-сервіс;
2. Чим відрізняються REST-сервіси від web-сервісів;
3. Три кроки до RESTFull сервісу.
4. Зв'язок REST-сервісу з протоколом HTTP.
5. Які рекомендації існують для REST сервісів з приводу URI.
6. Які рекомендації існують для REST сервісів з приводу пагінації і сортування.
7. Які рекомендації існують з приводу обробки виключень в REST-сервісах.
8. Які типи вхідних і вихідних запитів підтримуються в REST-сервісах і яким чином їх можна задати.
9. Які є анотації Jersey.
10. Чим анотація @QueryParam відрізняється від анотації @FormParam.

5.7 Завдання до захисту лабораторної роботи

1. Реалізувати і викликати з головної сторінки (сторінка для доменного об’єкта зі сторони «один») сервіс пошуку доменних об’єктів зі сторони один по будь-якому полю з використанням запитів Native SQL.
2. Реалізувати і викликати з головної сторінки (сторінка для доменного об’єкта зі сторони «один») сервіс пошуку доменних об’єктів зі сторони один по будь-якому полю з використанням запитів за критерієм.
3. Реалізувати і викликати з головної сторінки (сторінка для доменного об’єкта зі сторони «один») сервіс пошуку доменних об’єктів зі сторони один по будь-якому полю з використанням запитів HQL.
4. Реалізувати і викликати сервіс, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів Native SQL. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони «один».
5. Реалізувати і викликати сервіс, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів за критерієм. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони «один».
6. Реалізувати і викликати сервіс, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів HQL. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони «один».
7. Додати серверну валідацію (зверніть увагу, що валідація повинна бути саме на стороні сервера а не клієнта) для полів вводу для доменного об’єкту зі сторони «один» при доданні нового об’єкта. Виводити помилку валідації біля поля вводу, якщо воно містить помилку.
8. При додаванні, видаленні і заміні доменних об’єктів передавати параметри не через URI, а через тіло запиту (payload).
9. Реалізувати тег користувача, що звертається з використанням HTTP клієнта до REST-сервісу по отриманню доменних об’єктів, отримує список об’єктів і виводить їх на екран у вигляді таблички.
10. Реалізувати авторизацію і аутентифікацію для REST-сервісів, створених в рамках лабораторної роботи.

6 ЛАБОРАТОРНА РОБОТА №6
ВИВЧЕННЯ ТЕХНОЛОГІЙ ДЛЯ ІНТЕГРАЦІЇ ЗАСТОСУНКІВ

6.1 Мета роботи

Розглянути Java технології інтеграції застосунків, отримати практичні навички по використанню технологій інтеграції застосунків.

6.2 Теоретичні відомості

6.2.1 Основи web-сервісів

Web-сервіси перетворюють XML-документи в IT-системах. Web-сервіси – це XML-застосунки, що здійснюють зв’язування даних з програмами, об’єктами, базами даних або з бізнес операціями в цілому. Між web-сервісом і програмою здійснюється обмін XML-документами, що оформлені у вигляді повідомлень. Стандарти web-сервісів визначають формат таких повідомлень, інтерфейс, якому передається повідомлення, правила прив’язки вмісту повідомлення до реалізую чого сервіс застосунку і в зворотному напрямку, а також механізми публікації і пошуку інтерфейсів. Web-сервіси можуть використовуватися в багатьох застосунках. Незалежно від того, звідки завантажуються web-сервіси, з настільних комп’ютерів клієнтів чи з переносних, вони можуть використовуватися для звернення до таких internet-застосунків, як система попередніх замовлень або контролю виконання замовлень. Web-сервіси можна використовувати B2B-інтеграції (business-to-business), об’єднуючи застосунки, виконувані різними організаціями в один виробничий процес. Web-сервіси також можуть вирішувати більш широку проблему інтеграції застосунків підприємства (Enterprise Application Integration, EAI), здійснюючи зв'язок декількох застосунків одного підприємства з декількома іншими застосунками, що розміщені як до так і після брандмауера.

У всіх перелічених випадках технології web-сервісів є об’єднуючим елементом для різних частин програмного забезпечення. Як видно з рисунку 6.1, web-сервіси являють собою обгортку, що забезпечує стандартний спосіб взаємодії з прикладними програмними засобами такими як системи управління базами даних (СУБД), .NET, J2EE (Java2 Platform, Enterprise Edition), CORBA (Common Object Request Broker Architecture), посередники пакетів планування ресурсів підприємства (Enterprise Resource Planning, ERP), брокерів інтеграції та інших.

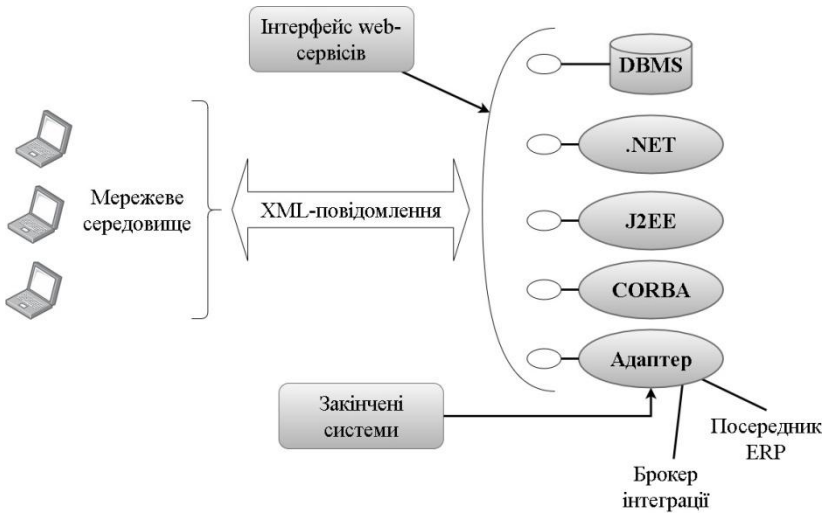

Рисунок 6.1 - Web-сервіси взаємодіють з прикладними системами

Інтерфейси web-сервісів отримують з мережевого середовища стандартні XML-повідомлення, перетворюють XML-дані в формат, який розуміє конкретна прикладна програмна система і відправляють повідомлення-відповідь (не є обов'язковим). Програмна реалізація web-сервісів (базове програмне забезпечення, нижній рівень) може бути створене на будь-якій мові програмування з використанням будь-якої операційної системи і дуб-якого зв'язного програмного забезпечення (middleware). Web-сервіси об'єднують програмування і концепції мережі. Web-сервіси поєднують параметри програмних застосунків і абстрактні характеристики мережі. Сучасні internet-технології частково досягають свої цілей, оскільки вони визначені на дуже високому рівні, що забезпечує сумісність з будь-якою операційною системою, будь-яким програмним і апаратним забезпеченням. Інфраструктура, що базується на застосуванні web-сервісів, користується цим рівнем абстракції і включає в себе зв'язану за даними семантичну інформацію, тобто web-сервіси визначають не тільки дані, але й порядок обробки і перетворення цих даних в базові програмні застосунки і в зворотному порядку.

6.2.2 Основні положення моделі web-сервісів

Концепція web-сервісів реалізується з допомогою ряду технологій, які стандартизовані World Wide Web Consortium (W3C). Взаємозв'язок цих технологій представлений на рисунку 6.2.

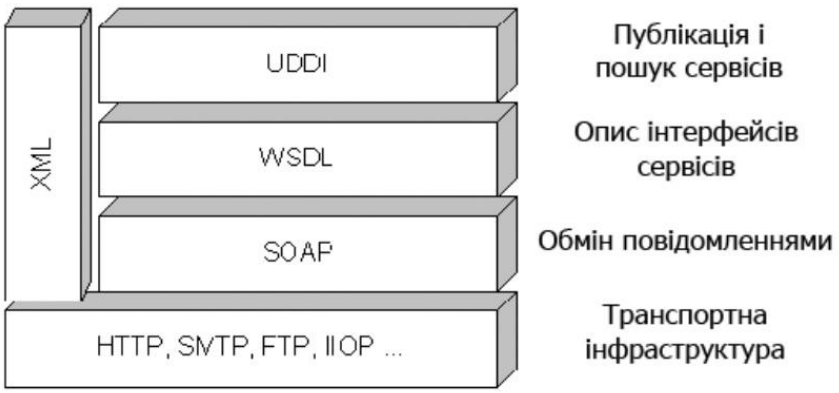

Рисунок 6.2 – Взаємозв'язок технологій web-сервісів

Web-сервіси є одним з варіантів реалізації компонентної архітектури. XML є фундаментом для створення більшості технологій, пов'язаних з web-сервісами. Для віддаленої взаємодії з web-сервісами використовується Simple Object Access Protocol (SOAP). SOAP забезпечує взаємодію розподілених систем незалежно від об'єктної моделі, операційної системи чи мови програмування. Дані визначаються у вигляді особливих XML документів особливого формату.

Відповідно до визначення W3C, web-сервіси це застосунки, які доступні по протоколам, які ж стандартними для мережі Internet. Нема вимог, щоб web-сервіси використовували який-небудь визначений транспортний протокол. Специфікація SOAP визначає, яким чином пов'язуються повідомлення SOAP і транспортний протокол. Повідомлення SOAP є звичайним XML документом. Цей документ складається з наступних XML-елементів:

- SOAP envelope, який визначає вміст повідомлення;
- SOAP header, в ньому міститься заготовочна інформація (не є обов'язковим елементом);
- SOAP body, в ньому міститься інформація викликів і відповідей на виклики.

Приклад SOAP повідомлення приведений нижче:

```
<soap: Envelope>
<soap: Body>
  <GetPrice>
  <Item>Apples</Item>
  </GetPrice>
</soap: Body>
</soap: Envelope>
```

Найбільш часто реалізується передача SOAP повідомлень по протоколу HTTP. Також широко поширене використання в якості транспортного протоколу SMTP, FTP, TCP.

Відповідно до визначення W3C WSDL – формат XML для опису мережевих сервісів як набору кінцевих операцій, що працюють з допомогою повідомлень, що містять документно-орієнтовану чи процедурно-орієнтовану інформацію. Документ WSDL цілком описує інтерфейс web-сервісу з зовнішнім світом. Він надає інформацію про послуги, які можна отримати, скориставшись методами сервісу, і способах звернення до цих методів.

Технологія Universal Description, Discovery and Integration (UDDI) передбачає ведення реєстру web-сервісів. Підключившись до цього реєстру, користувач може знайти web-сервіси, які найкращим чином задовольняють його потребам. Технологія UDDI дає можливість пошуку і публікації необхідного сервісу, як людиною, так і програмою-клієнтом. Пошук і публікація в реєстрі надається програмі-клієнту як набір web-сервісів реєстру UDDI.

Web-сервіси позиціонуються як програмне забезпечення проміжного шару. Використовувати web-сервіси можуть як клієнтські застосунки, що безпосередньо працюють з користувачем, так і інші застосунки (в тому числі і

інші web-сервіси). Концепція web-сервісів включає в себе можливість ведення реєстру web-сервісів. Опис інтерфейсу може бути отриманий з такого реєстру. Після створення і запровадження нового web-сервісу є сенс зареєструвати його в реєстрі. Тоді клієнти при пошуку сервісів, що реалізують вихідний інтерфейс, отримають вказівку на новий web-сервіс.

Web-сервіси можна використовувати як будівельні блоки при створенні застосунку. Застосунок може використовувати web-сервіси як віддалені компоненти, які надають певний функціонал. Існують різні сервіси, які надають якісне рішення таких задач як аутентифікація, ведення календаря, відправка повідомлень, пошук, переклад і т. д. Крім цього, можливі і інші варіанти використання web-сервісів. Наприклад, існують дослідження по використанню web-сервісів для побудови розподілених обчислень і інформаційних систем і однорангових і зі складною ієрархічною структурою.

6.2.3 Взаємодія web-сервісів

Web-сервіси підтримують декілька парадигм обміну повідомленнями. Рівень абстракції, на якому оперують web-сервіси, передбачає такі стилі взаємодії, як емуляцію віддаленого виклику процедури (Remote Procedure Call, RPC), асинхронний обмін повідомленнями, одностороннє відправлення повідомлення, широкомовлення і публікацію/підписку. Основні СУБД, такі як Oracle, SQL Server і DB2, підтримують аналіз XML і служби перетворення, що забезпечують безпосередню взаємодію між web-сервісом і СУБД.

Виробники програмного забезпечення зазвичай також надають можливість прив’язки web-сервісів до свої програмних систем (серверам застосунків і брокерам інтеграції). Відповідно, для користувача взаємодія з web-сервісом може проявлятися в інтерактивній або пакетній формі, що підтримує синхронну або асинхронну моделі зв’язку, а також як інтерфейс користувача, написаний з використанням Java, C#, офісних застосунків, браузерів чи «товстих» клієнтів СУБД. Така взаємодія може прив’язуватися до будь-якого типу базовій (більш низького рівня) програмній системі.

Web-сервіси виконують RPC- і документно-орієнтовану взаємодію. Стандарти і технології web-сервісів зазвичай включають два основних типи моделей взаємодії застосунків:

- віддалений виклик процедури (онлайнова);
- документно-орієнтований (пакетна).

Розглянемо кожен з типів окремо.

RPC-орієнтовані взаємодії зручні для короткого обміну даними. В RPC-орієнтованій взаємодії запити web-сервісів набувають форму виклику методи чи процедури з відповідними вхідними і вихідними параметрами. На відміну від документно-орієнтованої взаємодії RPC-орієнтована взаємодія виконує відправку документу, спеціально відформатованого для передачі в окрему логічну програму чи базу даних.

Документно-орієнтовані взаємодії зручні для обміну великими об’ємами даних. При документно-орієнтованій взаємодії запити до web-сервісу мають форму завершеного XML документу, призначено для обробки в цілому.

6.2.4 Технологія web-сервісів

Порядок опису, пошуку і взаємодії web-сервісів один з одним визначають стандарти. Взаємодіючі через Internet програми повинні вміти знаходити одна одну, знаходити інформацію, яка дозволяє здійснити зв'язок, розуміти, яка модель контактування повинна бути застосована (проста типу «запит/відповідь», чи більш складна послідовність), і домовитися про використання таких послуг, як захист інформації, підтвердження передачі повідомлень і складання угод. Деякі з цих сервісів реалізуються існуючими технологіями і пропонованими стандартами, а інші – ні. Спільнота, яка використовує web-сервіси намагається вдовольнити всі ці вимоги, але це – еволюційний процес, як і сама мережа Internet. З самого початку інфраструктура і стандарти web-сервісів передбачала можливість розширення (так як і до них XML та HTML), що дозволяє використовувати їх одразу після появи нових стандартів і технологій.

Для транспортування і перетворення даних в програми і в зворотному порядку web-сервіси потребують використання декількох суміжних XML технологій.

Мова XML (Extensible Markup Language) – фундамент, на якому будуються web-сервіси. Він надає мову визначення даних і порядок їх обробки. XML є сімейством пов’язаних специфікацій, тих, що публікуються і підтримуються internet-консорціумом (World Wide Web Consortium, W3C) і іншими організаціями.

WSDL (Web Services Description Language) – технологія, заснована на XML, що визначає інтерфейси web-сервісів, типи даних повідомлень, а також моделі взаємодії і протоколи зв’язування.

6.2.5 Анотації JAX-WS

До анотацій JAX-WS відносяться наступні анотації @WebService, @WebMethod, @SOAPBinding, @WebResult, @RequestWrapper, @ResponseWrapper, @Oneway. Розглянемо кожну з анотацій і приклади їх застосування.

Анотація @WebService застосовується для того, щоб вказати, що даний клас є web-сервісом. При створенні web-сервісу JAX-WS зазвичай створюється інтерфейс і реалізація інтерфейсу, які містять дану анотацію, але в інтерфейсі і реалізації інтерфейсу вона виконує різне призначення. Приклад застосування даної анотації приведений нижче:

```
//Приклад інтерфейсу
package ua.cn.stu.webservice;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
```

```
public interface WSAnnotationWebServiceI {
  @WebMethod
  float celsiusToFarhenheit(float celsius);
}

//Приклад реалізації інтерфейсу
package ua.cn.stu.webservice;

import javax.jws.WebService;

@WebService(endpointInterface="ua.cn.stu.webservice.WSAnnotationWebServiceI")
public class WsAnnotationsWebServiceImpl implements WSAnnotationWebServiceI {
  @Override
  public float celsiusToFarhenheit(float celsius) {
    return ((celsius - 32)*5)/9;
  }
}
```

Анотація @WebMethod може бути використана тільки для маркування методів і вказує, що метод репрезентує web-метод. Зазвичай анотація використовується в інтерфейсі web-сервісу. Приклад використання анотації приведений нижче:

```
package ua.cn.stu.webservice;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
public interface WSAnnotationWebServiceI {
  @WebMethod
  float celsiusToFarhenheit(float celsius);
}
```

Анотація @SOAPBinding використовується для вказування стилю SOAP повідомлення з допомогою атрибуту style, кий може приймати значення RPC і DOCUMENT. Іншим важливим призначенням даної анотації є вказування тип форматування SOPA повідомлення з допомогою атрибуту use, який може приймати значення LITERAL або ENCODED. Приклад використання даної анотації приведений нижче:

```
package ua.cn.stu.webservice;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL)
public interface WSAnnotationWebServiceI {
```

```
  @WebMethod
  float celsiusToFarhenheit(float celsius);
}
```

Анотація @WebResult використовується для вказування назв змінних в WSDL файлі. Приклад використання даної анотації показаний нижче:

```
package ua.cn.stu.webresult;

import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style = Style.RPC)
public interface WSAnnotationsWebResultI {
  @WebMethod
  @WebResult(partName="farhenheitResponse")
  float celsiusToFarhenheit(float celsius);
}
```

Анотація @RequestWrapper використовується для позначення методів в інтерфейсі web-сервісу та дає можливість обгорнути запит в обгортку. Дана анотація має чотири атрибути, які можна використовувати для налаштування обгортки: className, localName, partName, targetNamespace. Приклад використання даної анотації приведений нижче:

```
package ua.cn.stu.wrapper;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.xml.ws.RequestWrapper;

@WebService
@SOAPBinding(style=Style.RPC)
public interface WSRequestWrapperInterface {
  @WebMethod
  @RequestWrapper(localName="CTF",
  targetNamespace="http://stu.cn.ua/tempUtil",
  className="ua.cn.stu.webservice.CTF")
  float celsiusToFarhenheit(float celsius);
}
```

Анотація @ResponseWrapper використовується для позначення методів в інтерфейсі web-сервісу та дає можливість обгорнути відгук в обгортку. Дана анотація має чотири атрибути, які можна використовувати для налаштування обгортки: className, localName, partName, targetNamespace. Приклад використання даної анотації приведений нижче:

```
package ua.cn.stu.wrapper;
```

```
import javax.jws.WebMethod;
import javax.xml.ws.ResponseWrapper;

public interface WSResponseWrapperInterfaceI {
  @WebMethod
  @ResponseWrapper(localName="CTFResponse",
  targetNamespace="http://stu.cn.ua/tempUtil",
  className="ua.cn.stu.webservice.CTFResponse")
  float celsiusToFarhenheit(float celsius);
}
```

Анотація @Oneway – використовується для позначення web-методів і вказує, що web-метод буд використовуватися тільки для запиту та не буде повертати відповіді. Приклад використання даної анотації представлений нижче:

```
package ua.cn.stu.oneway;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService
@SOAPBinding(style = Style.RPC)
public interface WSAnnotationsOnewayI {
  @WebMethod
  @Oneway
  void sayHello();
}
```

6.3 Порядок виконання роботи

В даній лабораторній роботі в якості прикладу буде розглянуто процес створення набору web-сервісів XML (JAX-WS), з допомогою якого можна отримувати записи з бази даних, додавати нові записи, видаляти і оновлювати існуючі записи. В якості прикладу, як і в попередніх лабораторних роботах, був обраний об’єкт предметної області «інтернет-магазин» – продукт. Для створення набору web-сервсів використовується технологія Apache CXF. В якості web-сервера використовується web-сервер Apache Tomcat 7.0. В якості клієнта для перевірки працездатності набору сервісів буде використовуватися застосунок Soap UI, який необхідно попередньо завантажити і встановити.

1. Відкрийте IDE Eclipse і створіть новий Dynamic Web Project.
2. Виконайте конфігурацію вкладки Servers, якщо це не було зроблено в попередніх лабораторних роботах (див. кроки 1, 3 та 4 в порядку виконання лабораторної роботи №4).
3. Виконайте конфігурацію проекту для завантаження його на сервері Apache Tomcat. У вкладці Servers натисніть правою кнопкою миші на елементі

дерева Tomcat v7.0 Server at localhost і в контекстному меню натисніть на елемент Add and Remove. В діалоговому вікні перемістіть створений проект з поля Available в поле Configured, а всі інші проекти, що знаходяться в полі Configured перемістіть в поле Available , як показано на рисунку 6.3. Натисніть кнопку Finish.

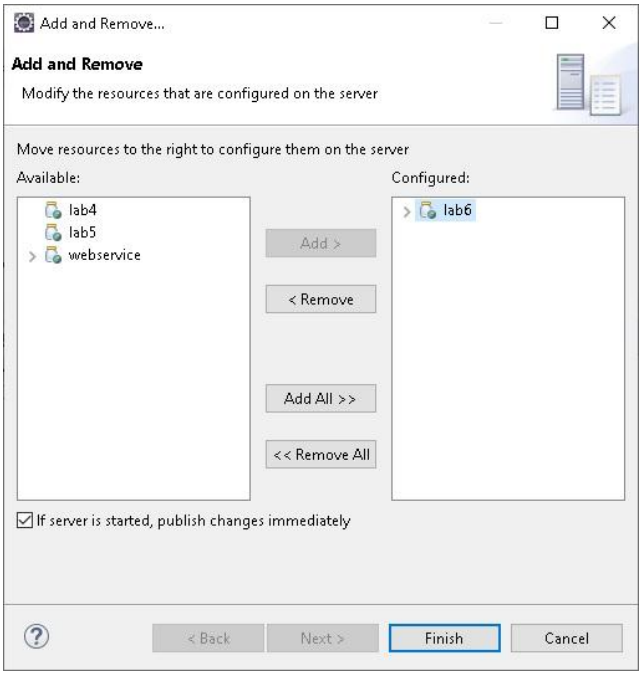

Рисунок 6.3 – Конфігурація проекту для завантаження на сервері

4. В створеному проекті Dynamic Web Project відкрийте WebContent/WEB-INF та перевірте в даному каталозі наявність каталогу lib. Якщо, даного каталогу нема, то створіть його.

5. Перевірте наявність в каталозі WebContent/WEB-INF конфігураційного файлу web.xml. Якщо файл відсутній, натисніть правою кнопкою миші на проекті та в контекстному меню оберіть пункти Java EE Tools -> Generate Deployment Descriptor Stub. Відкрийте файл та перегляньте його вміст. Даний конфігураційних файл містить налаштування web-застосунку, які ми будемо змінювати в процесі створення нашого застосунку.

6. Додайте в корінь проекту файл build.gradle. Даний файл буде використовуватися для завантаження всіх необхідних для застосунку бібліотек з використанням системи автоматизованого збирання Gradle. Для виконання даного файлу необхідно попередньо встановити систему автоматизованого збирання Gradle, як показано в лабораторній роботі №2. Контент файлу build.gradle представлений нижче:

```
apply plugin: 'war'

repositories {
  mavenCentral()
}

sourceSets {
  main {
```

```
    java {
      srcDir 'src'
    }
    java.outputDir = file('WebContent/WEB-INF/classes')
  }
}

dependencies {
  compile group: 'com.sun.xml.ws', name: 'jaxws-ri', version: '2.3.0'
  configurations.compile.exclude(group: 'com.sun.xml.ws',
    module: 'sdo-eclipselink-plugin')
  compile group: 'com.sun.activation', name: 'javax.activation',
    version: '1.2.0'
  compile group: 'javax.servlet', name: 'javax.servlet-api', version: '3.0.1'
  compile group: 'javax.servlet.jsp', name: 'jsp-api', version: '2.1'
  compile group: 'postgresql', name: 'postgresql', version: '8.3-603.jdbc4'
  compile group: 'org.hibernate', name: 'hibernate-core',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-entitymanager',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-osgi',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-envers',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-c3p0',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-proxool',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-infinispan',
    version: '4.3.6.Final'
  compile group: 'org.hibernate', name: 'hibernate-ehcache',
    version: '4.3.6.Final'
  compile group: 'antlr', name: 'antlr', version: '2.7.6'
  compile group: 'commons-collections', name: 'commons-collections',
    version: '3.1'
  compile group: 'dom4j', name: 'dom4j', version: '1.6.1'
  compile group: 'javassist', name: 'javassist', version: '3.4.GA'
  compile group: 'javax.transaction', name: 'jta', version: '1.1'
  compile group: 'org.slf4j', name: 'slf4j-api', version: '1.5.6'
  compile group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.5.6'
  compile group: 'org.apache.cxf', name: 'cxf-rt-features-logging',
    version: '3.3.0'
  compile group: 'org.apache.cxf', name: 'cxf-rt-frontend-simple',
    version: '3.3.0'
  compile group: 'org.apache.cxf', name: 'cxf-rt-transports-http',
    version: '3.3.0'
  compile group: 'org.apache.cxf', name: 'cxf-rt-frontend-jaxws',
    version: '3.3.0'
  compile group: 'org.apache.cxf', name: 'cxf-rt-transports-http-jetty',
    version: '3.3.0'
```

```
  compile group: 'org.springframework', name: 'spring-core',
    version: '3.2.8.RELEASE'
  compile group: 'org.springframework', name: 'spring-web',
    version: '3.2.8.RELEASE'
}

task copyToLib(type: Copy) {
  from configurations.runtime
  into "WebContent/WEB-INF/lib"
}

project.afterEvaluate {
  tasks.build.dependsOn(copyToLib)
}
```

7. Відкрийте консоль, перейдіть до кореня проекту і виконайте команду gradle build.

8. Поновіть каталог WebContent/WEB-INF/lib і переконайтеся, що після виконання команди gradle build в ньому з'явилися бібліотеки.

9. Підключіть всі бібліотеки з каталогу WebContent/WEB-INF/lib в CLASSPATH проекту. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Libraries натисніть кнопку Add JARs та додайте всі бібліотеки з каталогу WebContent/WEB-INF/lib. Натисніть кнопку Apply and close.

10. Задайте каталог, куди будуть компілюватися класи проекту. Він також, як і каталог lib, повинен знаходитися в каталозі WebContent/WEB-INF. Натисніть правою кнопкою миші на проект та в контекстному меню оберіть Build Path -> Configure Build Path. В діалоговому вікні на вкладці Source в полі вводу Default output folder вкажіть lab6/WebContent/WEB-INF/classes (lab6 – це ім'я проекту і, якщо у вас інше ім'я проекту, то цю частину необхідно замінити на ім'я вашого проекту). Натисніть кнопку Apply and close.

11. В каталозі src створіть пакети ua.cn.stu.dao та ua.cn.stu.domain. Перенесіть в пакет ua.cn.stu.dao класи HibernateDAOFactory і ProductDAO, а в пакет ua.cn.stu.domain клас Product, що використовувалися в попередніх лабораторних роботах (лабораторних роботах № 4 і №5).

12. В каталозі src створіть пакет ua.cn.stu.listener. Перенесіть в нього клас StartStopListener, що використовувався в попередніх лабораторних роботах №4 і №5.

13. Додайте налаштування для класу-слухача, створеного на попередньому кроці в конфігураційний файл web.xml. Конфігураційний тег, приведений нижче, повинен знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<listener>
  <listener-class>ua.cn.stu.listener.StartStopListener</listener-class>
</listener>
```

14. В каталозі src створіть пакет ua.cn.stu.service.

15. Створіть в пакеті ua.cn.stu.service інтерфейс з іменем ProductService і кодом як показано нижче:

```
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import ua.cn.stu.domain.Product;

@WebService
public interface ProductService {

  @WebMethod
  @WebResult(name = "Product")
  List<Product> getAllProducts();

  @WebMethod
  @WebResult(name = "Product")
  public Product addProduct(@WebParam(name = "Product") Product product);

  @WebMethod
  @WebResult(name = "Product")
  public Product updateProduct(@WebParam(name = "Product") Product product);

  @WebMethod
  @WebResult(name = "Product")
  public void deleteProduct(@WebParam(name = "ProductId") Long productId);

}
```

16. Створіть в пакеті ua.cn.stu.service клас з іменем ProductServiceImpl, що імплементує інтерфейс ProductService і має код як показано нижче:

```
import java.util.List;
import javax.jws.WebService;
import ua.cn.stu.dao.HibernateDAOFactory;
import ua.cn.stu.domain.Product;

@WebService(endpointInterface = "ua.cn.stu.service.ProductService",
  serviceName = "productservice")
public class ProductServiceImpl implements ProductService {

  @Override
  public List<Product> getAllProducts() {
    return HibernateDAOFactory.getInstance().getProductDAO().getAllProducts();
  }

  @Override
  public Product addProduct(Product product) {
    HibernateDAOFactory.getInstance().getProductDAO().createProduct(product);
    return product;
```

```
  }

  @Override
  public Product updateProduct(Product product) {
    HibernateDAOFactory.getInstance().getProductDAO().updateProduct(product);
    return product;
  }

  @Override
  public void deleteProduct(Long productId) {
    HibernateDAOFactory.getInstance().getProductDAO()
      .deleteProductById(productId);
  }

}
```

17. Додайте до каталогу WebContent/WEB-INF конфігураційний файл з іменем beans.xml, що використовується для налаштування web-сервісу:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint id="productservice"
    implementor="ua.cn.stu.service.ProductServiceImpl"
    address="/productservice" />

</beans>
```

18. Додайте до файлу web.xml конфігурацію контексту для підключення конфігураційного файлу beans.xml до дескриптору розгортання. Конфігураційний тег, приведений нижче, повинен знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/beans.xml</param-value>
</context-param>
```

19. Додайте до файлу web.xml конфігурацію контексту для ініціалізації контексту під час завантаження сервера. Конфігураційний тег, приведений нижче, повинен знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

20. Додайте до файлу web.xml конфігурацію для сервлета Apache CXF. Конфігураційні теги, приведені нижче, повинні знаходитись всередині тегу <web-app> на одному рівні з тегом <welcome-file-list>:

```
<servlet>
  <description>Apache CXF Endpoint</description>
  <display-name>cxf</display-name>
  <servlet-name>cxf</servlet-name>
  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>cxf</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

21. В закладці Servers натисніть правою кнопкою миші на елемент Tomcat v7.0 Server at localhost та натисніть на елемент меню Debug. Дочекайтеся завершення старту сервера, відкрийте браузер та в рядку запиту введіть http://localhost:8080/lab6/services/productservice?wsdl, де lab6 це контекст web-застосунку і він може відрізнятися в залежності від імені проекту. В браузері ви повинні побачити WSDL файл, що містить повний опис створеного набору web-сервісів.

22. Створіть в корені проекту каталог wsdl і збережіть в цей каталог WSDL файл, відкритий в браузері (для браузеру Chrome натисніть правою кнопкою миші на відкритий файл і в меню оберіть пункт меню Save as…, оберіть, створений каталог та натисніть кнопку Save). Поновіть каталог wsdl, вибравши його і натиснувши клавішу F5. Перегляньте збережений файл.

23. Файл буде збережений з розширенням .xml, але для WSDL файлів характерним є розширення .wsdl. Необхідно для зручності змінити розширення збереженого файлу. Натисніть на збережений файл правою кнопкою миші і в контекстному меню оберіть Refactor -> Rename. Змініть розширення файлу з .xml на .wsdl і натисніть кнопку OK. Збережений файл можна використовувати для автоматичної генерації набору класів для роботи зі створеним сервісом, що фактично відноситься до коду клієнта, але в даній лабораторній роботі генерація коду клієнта виконуватися не буде, а для валідації працездатності набору сервісів буде використовуватися застосунок Soap UI.

24. Завантажте з офіційного сайту https://www.soapui.org/ і встановіть застосунок Soap UI. Відкрийте застосунок Soap UI.

25. В головному меню застосунку Soap UI оберіть в головному меню пункти File -> New SOAP Project. В контекстному меню в полі Initial WSDL оберіть створений на попередніх кроках WSDL файл та натисніть кнопку OK. Проект повинен виглядати, як показано на рисунку 6.4. Методів в проекті повинні відповідати методам в інтерфейсі ProductService.

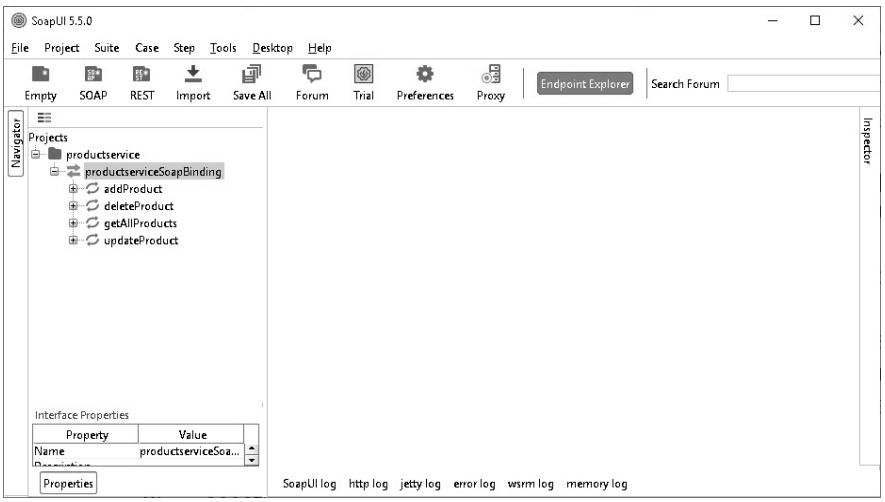

Рисунок 6.4 – Soap UI проект, створений з використанням збереженого WSDL файлу

26. Протестуйте набір сервісів з використанням застосунку Soap UI. Розкрите вузол дерева addProduct і натисніть два рази на елемент дерева Request1. У вікні Request 1 видаліть елемент id, ведіть значення для елементів description і name і виконайте запит. Вікно проекту перед виконанням запиту повинно виглядати приблизно як на малюнку 6.5. Елемент id необхідно видалити, оскільки в проекті реалізована автоматична генерація id.

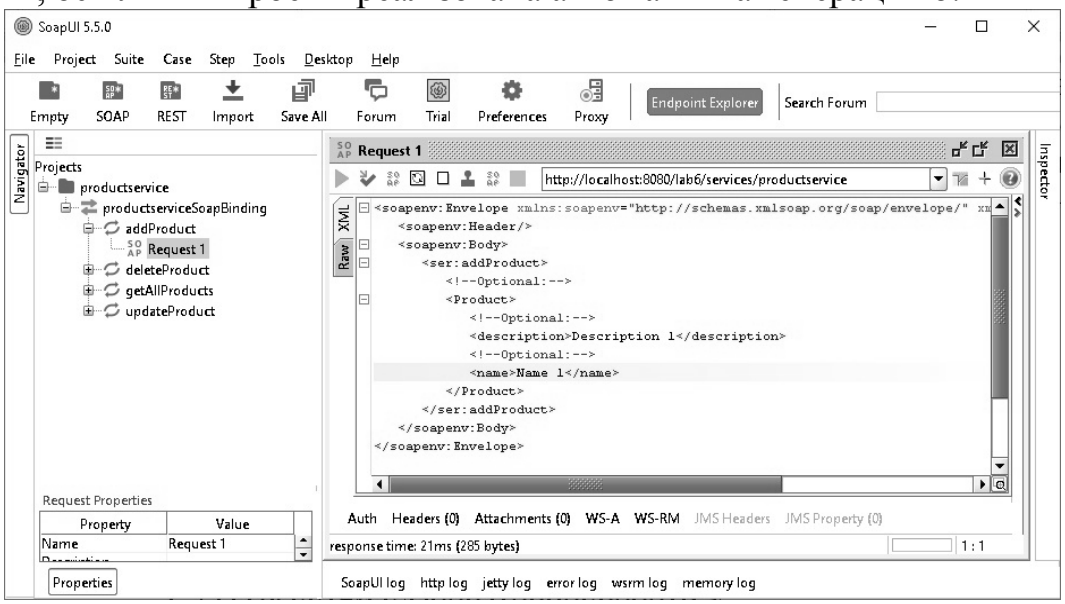

Рисунок 6.5 – Вікно проекту перед виконанням запиту

27. Розкрийте елемент дерева getAllProducts і натисніть два рази на елемент Request 1, що знаходиться під елементом getAllProducts. Виконайте запит на отримання списку всіх продуктів.

28. Протестуйте методи updateProduct і deleteproduct, використовуючи підходи описані на попередніх кроках. Для тестування методу updateProduct елемент id видаляти вже не треба.

6.4 Завдання на лабораторну роботу

Вдосконалити набір сервісів, описаний в ході виконання лабораторної роботи, щоб він міг працювати з двома сутностями, на відміну від набору

сервісів, описаного в лабораторній роботі. Предметну область необхідно вибрати з таблиці 6.1. Номер варіанта визначається по останній цифрі залікової книжки. Сутності повинні бути пов’язані відношенням один до багатьох. Зв'язок також повинен бути присутній на рівні створених сервісів.

Таблиця 6.1. Варіанти завдань

Номер варіанту	Предметна область
1	Піцерія (Офіціант, Відвідувач)
2	Крамниця (Постачальник, Товар)
3	Супермаркет (Продавець, Товар)
4	Відділення поліції (Поліцейський, Порушник)
5	Лікарня (Лікар, Пацієнт)
6	Кінотеатр (Кінозал, Відвідувач)
7	Корабель (Каюта, Пасажир)
8	Підприємство (Відділ, Співробітник)
9	Бібліотека (Книга, Читач)

6.5 Що повинно бути у звіті

1. Назва і тема лабораторної роботи;
2. Мета лабораторної роботи;
3. Короткі теоретичні відомості (не більше але і не менше ніж на одну сторінку);
4. Хід виконання лабораторної роботи;
5. Тексти програм;
6. Висновки.

6.6 Контрольні питання

1. Які є методи інтеграції застосунків.
2. Що таке web-сервіс XML.
3. Що таке SOAP.
4. Структура SOAP пакету.
5. Що таке WSDL.
6. Яка структура WSDL файлу.
7. Що таке UDDI.
8. Які є анотації Apache CXF.
9. Які є способи взаємодії web-сервісів.
10. Що таке RPC-орієнтована взаємодія.

6.7 Завдання до захисту лабораторної роботи

1. Реалізувати web-сервіс XML для пошуку доменних об’єктів зі сторони один по будь-якому полю з використанням запитів Native SQL.
2. Реалізувати web-сервіс XML для пошуку доменних об’єктів зі сторони один по будь-якому полю з використанням запитів за критерієм.
3. Реалізувати web-сервіс XML для пошуку доменних об’єктів зі

сторони один по будь-якому полю з використанням запитів HQL.

4. Реалізувати і викликати сервіс, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів Native SQL. TOP 10 об’єктів повинно відображатися на JSP сторінці доменного об’єкта зі сторони «один».

5. Реалізувати web-сервіс XML, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів за критерієм.

6. Реалізувати web-сервіс XML, що отримує TOP 10 доменних об’єктів зі сторони багато з використання запитів HQL.

7. Реалізувати web-сервіс XML, що повертає список об’єктів зі сторони один посторінково. Номер початкового об’єкт повинен задаватися з використанням параметру beginIndex, а кількість необхідних об’єктів повинне задаватися з використанням параметру count.

8. Реалізувати тег користувача, що звертається до web-сервісу XML для отримання доменних об’єкті і виводить їх на екран у вигляді таблички.

9. Реалізувати клієнт у вигляді настільного застосунку, що використовує створений набір сервісів для здійснення маніпуляцій з доменними об’єктами (отримання списку об’єктів, додання, видалення, заміна об’єктів).

10. Реалізувати логіку для здійснення авторизації і аутентифікації для створеного набору сервісів.

7 ЗАВДАННЯ ДО РОЗРАХУНКОВО-ГРАФІЧНОЇ РОБОТИ

7.1 Мета розрахункової роботи

Ознайомитися і отримати практичні навички по роботі з технологіями, що не увійшли в лекційний курс і курс лабораторних робіт, такими як Spring Data, Spring Web Flow, Spring Web Services, Apache Solr, Apache Active MQ, Sphinx та Elasticsearch.

7.2 Що повинно бути у звіті

1. Тема розрахунково-графічної роботи;
2. Мета розрахунково-графічної роботи;
3. Теоретичні відомості, пов’язані з темою розрахунково-графічної роботи;
4. Хід виконання розрахунково-графічної роботи;
5. Тексти програми;
6. Висновки.

7.3 Завдання до розрахунково-графічної роботи

Завдання обирається відповідно до вподобань здобувача освіти. Якщо здобувач не зміг обрати завдання на розрахунково-графічну роботу до кінця вказаного терміну, завдання призначається примусово із тих завдань, що залишились. Використання технології, вказаної в завданні є обов’язковим.

7.3.1 Завдання №1

Технологія Spring Data. Застосунок повинен взаємодіяти з No SQL базою даних Mongo DB та реалізовувати функції додання нових даних та пошуку існуючих даних. Предметна область – internet-крамниця (продукт).

7.3.2 Завдання №2

Технологія Spring Data. Застосунок повинен взаємодіяти з No SQL базою даних Mongo DB та реалізовувати функції додання нових даних та пошуку існуючих даних. Предметна область – кафе (відвідувач кафе).

7.3.3 Завдання №3

Технологія Spring Data. Застосунок повинен взаємодіяти з No SQL базою даних Mongo DB та реалізовувати функції додання нових даних та пошуку існуючих даних. Предметна область – лікарня (медична картка пацієнта).

7.3.4 Завдання №4

Технологія Spring Data. Застосунок повинен взаємодіяти з No SQL базою даних Mongo DB та реалізовувати функції додання нових даних та пошуку існуючих даних. Предметна область – бібліотека (читацький абонемент).

7.3.5 Завдання №5

Технологія Spring Data. Застосунок повинен взаємодіяти з No SQL базою даних Mongo DB та реалізовувати функції додавання нових даних та пошуку існуючих даних. Предметна область – система автоматизованої фіксації порушень ПДР (запис про порушення).

7.3.6 Завдання №6

Технологія Apache Solr. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Apache Solr. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Apache Solr. Пошук даних повинен здійснюватися з використанням Apache Solr. Предметна область – internet-крамниця (продукт).

7.3.7 Завдання № 7

Технологія Apache Solr. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Apache Solr. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Apache Solr. Пошук даних повинен здійснюватися з використанням Apache Solr. Предметна область – кафе (відвідувач кафе).

7.3.8 Завдання №8

Технологія Apache Solr. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Apache Solr. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Apache Solr. Пошук даних повинен здійснюватися з використанням Apache Solr. Предметна область – лікарня (медична картка пацієнта).

7.3.9 Завдання №9

Технологія Apache Solr. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Apache Solr. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Apache Solr. Пошук даних повинен здійснюватися з використанням Apache Solr. Предметна область – бібліотека (читацький абонемент).

7.3.10 Завдання №10

Технологія Apache Solr. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Apache Solr. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Apache Solr. Пошук даних повинен здійснюватися з використанням Apache Solr.

Предметна область – система автоматизованої фіксації порушень ПДР (запис про порушення).

7.3.11 Завдання №11

Технологія Elasticsearch. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Elasticsearch. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Elasticsearch. Пошук даних повинен здійснюватися з використанням Elasticsearch. Предметна область – internet-крамниця (продукт).

7.3.12 Завдання №12

Технологія Elasticsearch. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Elasticsearch. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Elasticsearch. Пошук даних повинен здійснюватися з використанням Elasticsearch. Предметна область – кафе (відвідувач кафе).

7.3.13 Завдання №13

Технологія Elasticsearch. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Elasticsearch. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Elasticsearch. Пошук даних повинен здійснюватися з використанням Elasticsearch. Предметна область – лікарня (медична картка пацієнта).

7.3.14 Завдання №14

Технологія Elasticsearch. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Elasticsearch. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Elasticsearch. Пошук даних повинен здійснюватися з використанням Elasticsearch. Предметна область – бібліотека (читацький абонемент).

7.3.15 Завдання №15

Технологія Elasticsearch. Застосунок повинен використовувати в якості джерел даних реляційну базу даних і Elasticsearch. Додання нових даних повинно здійснюватися в реляційну базу даних. Після додання нових даних в базу даних повинна бути передбачена процедура індексування даних в Elasticsearch. Пошук даних повинен здійснюватися з використанням Elasticsearch. Предметна область – система автоматизованої фіксації порушень ПДР (запис про порушення).

7.3.16 Завдання №16

Технологія Apache Active MQ. Необхідно створити два застосунки. Перший застосунок відправляє повідомлення в чергу, а другий застосунок зчитує повідомлення з черги і додає дані в базу даних. Повідомлення містить об’єкт «Продукт» предметної області «Internet-крамниця».

7.3.17 Завдання №17

Технологія Apache Active MQ. Необхідно створити два застосунки. Перший застосунок відправляє повідомлення в чергу, а другий застосунок зчитує повідомлення з черги і додає дані в базу даних. Повідомлення містить об’єкт «Відвідувач кафе» предметної області «Кафе».

7.3.18 Завдання №18

Технологія Apache Active MQ. Необхідно створити два застосунки. Перший застосунок відправляє повідомлення в чергу, а другий застосунок зчитує повідомлення з черги і додає дані в базу даних. Повідомлення містить об’єкт «Медична картка пацієнта» предметної області «Лікарня».

7.3.19 Завдання №19

Технологія Apache Active MQ. Необхідно створити два застосунки. Перший застосунок відправляє повідомлення в чергу, а другий застосунок зчитує повідомлення з черги і додає дані в базу даних. Повідомлення містить об’єкт «Читацький абонемент» предметної області «Бібліотека».

7.3.20 Завдання №20

Технологія Apache Active MQ. Необхідно створити два застосунки. Перший застосунок відправляє повідомлення в чергу, а другий застосунок зчитує повідомлення з черги і додає дані в базу даних. Повідомлення містить об’єкт «Запис про порушення» предметної області «Система автоматизованої фіксації порушень ПДР».

7.3.21 Завдання № 21

Технологія Spring Web Flow. Необхідно створити web-застосунок з використанням технології Spring Web Flow, що дозволяє додавати, видаляти та переглядати дані про об’єкт «Продукт» предметної області «Internet-крамниця».

7.3.22 Завдання №22

Технологія Spring Web Flow. Необхідно створити web-застосунок з використанням технології Spring Web Flow, що дозволяє додавати, видаляти та переглядати дані про об’єкт «Відвідувач кафе» предметної області «Кафе».

7.3.23 Завдання № 23

Технологія Spring Web Flow. Необхідно створити web-застосунок з використанням технології Spring Web Flow, що дозволяє додавати, видаляти та

переглядати дані про об’єкт «Медична картка пацієнта» предметної області «Лікарня».

7.3.24 Завдання №24

Технологія Spring Web Flow. Необхідно створити web-застосунок з використанням технології Spring Web Flow, що дозволяє додавати, видаляти та переглядати дані про об’єкт «Читацький абонемент» предметної області «Бібліотека».

7.3.25 Завдання №25

Технологія Spring Web Flow. Необхідно створити web-застосунок з використанням технології Spring Web Flow, що дозволяє додавати, видаляти та переглядати дані про об’єкт «Запис про порушення» предметної області «Система автоматизованої фіксації порушень ПДР».

7.3.26 Завдання №26

Технологія Spring Web Services. Необхідно створити набір REST-сервісів з використанням технології Spring Web Services, які дозволяють додавати, видаляти та переглядати дані про об’єкт «Продукт» предметної області «Internet-крамниця».

7.3.27 Завдання №27

Технологія Spring Web Services. Необхідно створити набір REST-сервісів з використанням технології Spring Web Services, які дозволяють додавати, видаляти та переглядати дані про об’єкт «Відвідувач кафе» предметної області «Кафе».

7.3.28 Завдання №28

Технологія Spring Web Services. Необхідно створити набір REST-сервісів з використанням технології Spring Web Services, які дозволяють додавати, видаляти та переглядати дані про об’єкт «Медична картка пацієнта» предметної області «Лікарня».

7.3.29 Завдання №29

Технологія Spring Web Services. Необхідно створити набір REST-сервісів з використанням технології Spring Web Services, які дозволяють додавати, видаляти та переглядати дані про об’єкт «Читацький абонемент» предметної області «Бібліотека».

7.3.30 Завдання №30

Технологія Spring Web Services. Необхідно створити набір REST-сервісів з використанням технології Spring Web Services, які дозволяють додавати, видаляти та переглядати дані про об’єкт «Запис про порушення» предметної області «Система автоматизованої фіксації порушень ПДР».

7.3.31 Завдання №31

Технологія Sphinx. Створити застосунок, що здійснює пошук даних в джерелі даних з допомогою ключових слів. Ввід ключових слів забезпечити з використанням технології Sphinx. В якості джерела даних використати Apache Solr. Предметна область – internet-крамниця (продукт).

7.3.32 Завдання №32

Технологія Sphinx. Створити застосунок, що здійснює пошук даних в джерелі даних з допомогою ключових слів. Ввід ключових слів забезпечити з використанням технології Sphinx. В якості джерела даних використати Apache Solr. Предметна область – кафе (відвідувач кафе).

7.3.33 Завдання №33

Технологія Sphinx. Створити застосунок, що здійснює пошук даних в джерелі даних з допомогою ключових слів. Ввід ключових слів забезпечити з використанням технології Sphinx. В якості джерела даних використати Apache Solr. Предметна область – лікарня (медична картка пацієнта).

7.3.34 Завдання №34

Технологія Sphinx. Створити застосунок, що здійснює пошук даних в джерелі даних з допомогою ключових слів. Ввід ключових слів забезпечити з використанням технології Sphinx. В якості джерела даних використати Elasticsearch. Предметна область – бібліотека (читацький абонемент).

7.3.35 Завдання №35

Технологія Sphinx. Створити застосунок, що здійснює пошук даних в джерелі даних з допомогою ключових слів. Ввід ключових слів забезпечити з використанням технології Sphinx. В якості джерела даних використати Elasticsearch. Предметна область – система автоматизованої фіксації порушень ПДР (запис про порушення)..

8 ЗАВДАННЯ ДО САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ

Даний розділ містить посилання на підручники і навчальні посібники, які допоможуть самостійно опрацювати кожну з лекційних тем.

8.1 Тема 1. Платформа Java і її основні компоненти

1. Benjamin J Evans, James Gough, Chris Newland. Optimizing Java: Practical Techniques for Improving JVM Application Performance. O'Reilly, 2018.-440 p.

8.2 Тема 2. Технологія динамічного визначення типів

1. Ira R. Forman and Nate Forman. Java Reflection in Action. Manning, 2004.-300 p.

8.3 Тема 3. Технології модульного і функціонального тестування

1. Ira Sander Mak. Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications 1st Edition. O'Reilly, 2017.-300 p.
2. Tomek Kaczanowski. Practical Unit Testing with JUnit and Mockito. Tomasz Kaczanowski, 2013.-402 p.

8.4 Тема 4. Технології зв’язування XML даних

1. Brett McLaughlin. Java & XML, 2nd Edition: Solutions to Real-World Problems. O'Reilly, 2001.-550 p.

8.5 Тема 5. Технологія JDBC

1. George Reese. Java Database Best Practices. O'Reilly, 2003.-304 p.

8.6 Тема 6. Технології об’єктно-реляційного відображення

1. Christian Bauer, Gavin King. Java Persistence with Hibernate: Revised Edition of Hibernate in Action. Manning, 2006.-904 p.

8.7 Тема 7. Технології повнотекстового пошуку

1. Rafal Kuc. Apache Solr 3.1 Cookbook. Packt Publishing Limited, 2011.-300 p.
2. Roy Russo. Elasticsearch in Action. Manning, 2015.-496 p.

8.8 Тема 8. Технологія розпізнавання мови Sphinx

1. Kai-Fu Lee. Automatic Speech Recognition : The Development of the SPHINX System. Springer-Verlag, 2013.-207 p.

8.9 Тема 9. Java технології Servlet та JSP

1. Bruce W. Perry. Java Servlet & JSP Cookbook, 1st Edition. O'Reilly, 2003.-704 p.

8.10 Тема 10. Сервіси REST та web-сервіси XML

1. Bill Burke. RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services. O'Reilly, 2013.-392 p.
2. James Snell, Doug Tidwell, Pavel Kulchenko. Programming Web Services With SOAP. O'Reilly, 2001.-262 p.
3. Thomas Erl. Service-Oriented Architecture: Analysis and Design for Services and Microservices (2nd Edition) (The Pearson Service Technology Series from Thomas Erl). Prentice Hall, 2016.-416 p.

8.11 Тема 11. Технології інтегрування застосувань

1. Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld. Spring Integration in Action. Manning, 2012.-368 p.
2. Gregor Hohpe, Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2003.-736 p.

8.12 Тема 12. Підтримка MVC в web-застосунках

1. Henry H. Liu. Spring 4 for Developing Enterprise Applications: An End-to-End Approach. CreateSpace, 2012.-426 p.

8.13 Тема 13. Технології автоматизованого збирання проекту

1. Tim Berglund, Matthew McCullough. Building and Testing with Gradle. O'Reilly, 2011.-116 p.
2. Brian R Jackson. Maven: The Definitive Guide. O'Reilly, 2015.-350 p.

8.14 Тема 14. Технології обробки великих об’ємів даних

1. Chuck Lam. Hadoop in Action. Manning, 2010.-336 p.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Литвинов В. В., Казимир В.В., Пріла О. А., Харченко М. В., Задорожній А. О. Лабораторний практикум з об’єктно-орієнтованих технологій розробки прикладних програм. Навчальний посібник.–Чернігів: видавництво Чернігівського державного інституту економіки і управління, 2014.- 364 с.
2. Herbert Schildt. Java2:The Complete Reference, Fifth Edition. Osborne, 2002.- 1186 p.
3. Bruce W. Perry. Java Servlet & JSP Cookbook, 1st Edition. O'Reilly, 2003.- 704 p.
4. Benjamin Muschko. Gradle in Action. Manning, 2014.- 395 p.
5. Christian Bauer and Gavin King. Hibernate in Action. Manning, 2004.-400 p.
6. Robert C Martin. Clean Code : A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.- 464 p.
7. Brian Cole, Robert Eckstein. Java™ Swing, 2nd edition. O`Reilly, 2002. – 1278 p.
8. Система дистанційного навчання «moodle» [Електронний ресурс]. – Режим доступу : http://eln.stu.cn.ua/course/... Java та C# технології прикладного програмування.