**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информатики и вычислительной техники

Кафедра ИиСП

**Отчёт по курсовой работе по дисциплине**

**«Разработка интернет приложений»**

Выполнил:

студент

группы ПС-41

Колчин И.А.

Проверил:

Габдулин Д. И.

г. Йошкар-Ола

2017

Оглавление

[Задание 2](#_Toc503855405)

[Решение 2](#_Toc503855406)

[Структура программы 3](#_Toc503855407)

[Описание работы программы 4](#_Toc503855408)

[Набор тестов 5](#_Toc503855409)

[Ссылка на код лабораторных работ 6](#_Toc503855410)

[Листинг 6](#_Toc503855411)

[Использованные инструменты 29](#_Toc503855412)

# Задание

1. Выбрать предметную область (книги, машины, одежда)
2. Определить атрибуты для сущности предметной области (цвет, размер, название и т.д.)
3. Создать форму добавляющую новую сущность
4. Вывести список внесенных сущностей
5. Создать тесты для функционала
6. Организовать хранение сущностей в базе данных NuoDB
7. Реализовать простейший шаблонизатор
8. Вынести модель данных, бизнес логику и представление из Servlet’а
9. Добавить логирование событий и ошибок

# Решение

Предметной областью у нас будут книги.

Атрибуты книги: автор книги, название книги, дата выпуска и количество страниц в книге.

# Структура программы

В качестве предметной области были выбраны книги.

Программа состоит из следующих классов

* **MainServlet**

Обрабатывает запросы от пользователя, проверяет данные и генерирует страницы.

* **HtmlTemplater**

Шаблонизатор из 3 лабы, его задача генерировать страницу на основе шаблона и переданных переменных.

* **TemplateVariable**

Переменная для шаблонизатора, хранит имя и объект, который будет использоваться шаблонизатором.

* **BookDatabase**

Отвечает за работу с базой данных, позволяет добавлять, удалять данные.

* **Book**

В качестве предметной области были выбраны книги. Книга содержит в себе автора, название, количество страниц, дату издания.

При запуске приложения сначала открывается страница добавления книги. После загрузки страницы пользователь может добавить новую книгу или посмотреть список книг.

# Описание работы программы

Далее будет описано взаимодействие классов на примере действий пользователя и принцип работы шаблонизатора.

**Добавление книги:**

1. Отправка запроса с данными для новой книги классу **MainServlet**
2. Проверка данных
3. **MainServlet** формирует книгу(класс **Book**) и передаёт **BookDatabase** для добавления в базу
4. Если не все данные введены или введены не корректно то появится сообщение в верхней части окна. Для этого шаблонизатору(**HtmlTemplater**) будет дана команда сгенерировать страницу добавления книги с сообщением об ошибке. Перед генерацией шаблонизатору будет передано сообщение об ошибке, а затем дана команда для генерации страницы.
5. Если всё введено правильно то страница перезагрузится и вернётся в изначальное состояние. Шаблонизатор создаст новую страницу.

**Просмотр списка книг:**

1. Отправка запроса с данными для новой книги классу **MainServlet**
2. Сервлет запросит книги у **BookDatabase** и передаст шаблонизатору(**HtmlTemplater**).
3. Затем шаблонизатор сгенерирует страницу со списком

**Принцип работы шаблонизатора:**

1. Сначала передаются переменные, которые будут использоваться для генерации страницы с помощью команды **setAttribute(String name, String type, Object object),** где **name** – имя переменной, **type –**тип, **object**  - объект которые нужно передать.
2. Внутри шаблонизатора при выполнении **setAttribute** в список переменных вносится переменная в виде класса **TemplateVariable** куда передаются полученные аргументы.
3. Далее при генерации страницы название переменной используется для поиска в списке с **TemplateVariable**. Затем проверяется тип и производится конвертация содержимого переменной в строку для отображения.

# Набор тестов

Тесты для **MainServlet**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Имя** | **Автор** | **Кол-во страниц** | **Дата публикации** |
| **Положительные тесты** | **Тестирование поля для кол-ва страниц** | | | |
| n | a | 0 | 12.12.2012 |
| n | a | 2017 | 12.12.2012 |
| n | a | Integer.MAX\_VALUE | 12.12.2012 |
| **Тестирование поля для даты публикации** | | | |
| n | a | 100 | 12.12.2012 |
| **Тестирование поля для автора и названия книги** | | | |
| a | b | 1 | 12.12.2012 |
| **Отрицательные** | **Тестирование пустых полей** | | | |
|  | a | 5 | 12.12.2012 |
| n |  | 2 | 12.12.2012 |
| c | a |  | 12.12.2012 |
| c | a | 3 |  |
| **Тестирование поля для кол-ва страниц** | | | |
| n | a | -9999999999 | 12.12.2012 |
| n | a | 9999999999 | 12.12.2012 |
| **Тестирование поля для даты публикации** | | | |
| n | a | 100 | 120.12.2012 |
| n | a | 100 | 2012 |
| n | a | 100 | d |
| n | a | 100 |  |

Тесты для **HtmlTemplater** на извлечении переменных

|  |  |
| --- | --- |
| **Положительные тесты** | **Отрицательные** |
| $(name) | a |
| $(book.name) | aaa |
| <t>$(book.name)</t>\n$(var) | 12 |
|  | -12 |
|  | 0 |
|  | 0 |
|  | name |
|  | book.name |
|  | $name |
|  | $book.name |
|  | $ |
|  | $( |
|  | $() |

# Ссылка на код лабораторных работ

<https://github.com/7kia/PIA>

# Листинг

**Book.java**

**public** **class** Book {

**public** String name;

**public** String author;

**public** Integer pageAmount;

**public** Date publishingData;

**public** Book()

{

}

**public** Book(

String name,

String author,

Integer pageAmount,

Date publishingData

) {

**this**.name = name;

**this**.author = author;

**this**.pageAmount = pageAmount;

**this**.publishingData = publishingData;

}

**public** String getName() {

**return** name;

}

**public** String getAuthor() {

**return** author;

}

**public** Integer getPageAmount() {

**return** pageAmount;

}

**public** Date getPublishingData() {

**return** publishingData;

}

}

**BookDatabase.java**

public class BookDatabase {

public static final String DATABASE\_URL = "jdbc:com.nuodb://localhost/";

private final Connection dbConnection;

static private final String dbName = "books";

private static final Logger log = Logger.getLogger(MainServlet.class);

private ArrayList<String> columnNames;

private ArrayList<String> columnTypes;

public BookDatabase(

String user,

String password,

String dbName

) throws SQLException, ClassNotFoundException

{

Properties properties = new Properties();

properties.put("user", user);

properties.put("password", password);

properties.put("schema", "hello");

dbConnection = DbConnection.getConnection(DATABASE\_URL + dbName, properties);

columnNames = new ArrayList<String>();

columnTypes = new ArrayList<String>();

columnNames.add("name");

columnTypes.add("string");

columnNames.add("author");

columnTypes.add("string");

columnNames.add("pageAmount");

columnTypes.add("int");

columnNames.add("publishingData");

columnTypes.add("DATE");

}

public void close() throws SQLException {

dbConnection.close();

}

public void createTable() throws SQLTimeoutException {

try {

Statement stmt = dbConnection.createStatement();

stmt.execute("CREATE TABLE " + dbName

+ " (id int primary key, "+ getCreateColumnList() + ")"

);

dbConnection.commit();

} catch(SQLTimeoutException exception) {

log.error("SQLTimeoutException exception. Skipping table creation: " + exception.getMessage());

} catch(SQLException exception) {

log.error("SQLException, skipping table creation: " + exception.getMessage());

}

}

private String getCreateColumnList()

{

String result = "";

for(int i = 0; i < columnNames.size(); i++)

{

result += columnNames.get(i) + " " + columnTypes.get(i);

if((i + 1) < columnNames.size()) {

result += ", ";

}

}

return result;

}

private String generateNameColumnList()

{

String listValues = "";

for (int i = 0; i < columnNames.size(); i++) {

listValues += columnNames.get(i);

if((i + 1) < columnNames.size()) {

listValues += ", ";

}

}

return "id, " + listValues;

}

private String generateValuesPartStatement()

{

String listValues = "";

for (int i = 0; i <= columnNames.size(); i++) {

listValues += "?";

if((i + 1) <= columnNames.size()) {

listValues += ", ";

}

}

return " values (" + listValues + ")";

}

public void insertName(Book book, int id) throws SQLTimeoutException {

try {

PreparedStatement stmt = dbConnection.prepareStatement(

"insert into " + dbName +

" (" + generateNameColumnList() + ")" +

generateValuesPartStatement()

);

stmt.setInt(1, id);

stmt.setString(2, book.name);

stmt.setString(3, book.author);

stmt.setInt(4, book.pageAmount);

stmt.setDate(5, book.publishingData);

stmt.addBatch();

stmt.executeBatch();

dbConnection.commit();

} catch(SQLException exception) {

log.error("SQLException, Skipping insert: " + exception.getMessage());

} catch(Exception exception) {

log.error("Untracked exception. Skipping insert..." + exception.getMessage());

}

}

public String getName(int id) throws SQLException {

final PreparedStatement pst = dbConnection.prepareStatement(

getSelectStatement("name", dbName, "id=?")

);

pst.setInt(1, id);

ResultSet resultSet = pst.executeQuery();

if (resultSet.next()) {

return resultSet.getString(1);

}

resultSet.close();

return null;

}

public boolean isEmpty()

{

try {

PreparedStatement stmt = dbConnection.prepareStatement(

getSelectStatement("count(\*)", dbName, "id=?")

);

ResultSet resultSet = stmt.executeQuery();

if (resultSet.next()) {

return resultSet.getInt(1) == 0;

}

} catch(SQLException exception) {

return true;

} catch(Exception exception) {

log.error("Untracked exception. Skipping insert..." + exception.getMessage());

}

return false;

}

public void clear()

{

try {

PreparedStatement stmt = dbConnection.prepareStatement(

getDeleteStatement(dbName)

);

stmt.addBatch();

stmt.executeBatch();

dbConnection.commit();

} catch (SQLException exception) {

log.error("SQLException, Skipping clear: " + exception.getMessage());

}

}

private String getDeleteStatement(String tableName)

{

return "DELETE FROM " + dbName + " where id > 0 ";

}

public String getPublishingData(int id) throws SQLException {

PreparedStatement pst = dbConnection.prepareStatement(

getSelectStatement("publishingData", dbName, "id=?")

);

pst.setInt(1, id);

ResultSet resultSet = pst.executeQuery();

if (resultSet.next()) {

return resultSet.getString(1);

}

return null;

}

public Book getBook(int id) throws SQLException {

PreparedStatement pst = dbConnection.prepareStatement(

getSelectStatement("\*", dbName, "id=" + id)

);

Book book = new Book();

ResultSet resultSet = pst.executeQuery();

if (resultSet.next()) {

book.name = resultSet.getString(2);

book.author = resultSet.getString(3);

book.pageAmount = resultSet.getInt(4);

book.publishingData = resultSet.getDate(5);

}

return book;

}

public List<Book> getBooks() throws SQLException {

PreparedStatement pst = dbConnection.prepareStatement(

"SELECT MAX(id) FROM " + dbName

);

List<Book> books = new ArrayList<Book>();

ResultSet resultSet = pst.executeQuery();

if (resultSet.next()) {

int lastId = resultSet.getInt(1);

for(int i = 1; i <= lastId; i++) {

books.add(getBook(i));

}

}

return books;

}

private String getSelectStatement(

String columns,

String tableName,

String condition

) {

return "select " + columns + " from "

+ tableName + " where " + condition;

}

}

**DBConnection.java**

public class DbConnection {

public static final String DRIVER\_CLASS = "com.nuodb.jdbc.Driver";

public static Connection getConnection(String url, Properties info)

throws ClassNotFoundException

{

Connection connection = null;

try {

Class.forName(DRIVER\_CLASS);

return DriverManager.getConnection(url, info);

} catch (ClassNotFoundException | SQLException exception) {

exception.printStackTrace();

}

return connection;

}

}

**TemplateVariable.java**

**public** **class** TemplateVariable {

**public** String type;

**public** Object object;

**public** TemplateVariable(String type, Object object)

{

**this**.type = type;

**this**.object = object;

}

}

**MainServlet.java**

@WebServlet("/MainServlet")

public class MainServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

private String errorMessage = new String();

private BookDatabase bookDatabase;

private static final Logger log = Logger.getLogger(MainServlet.class);

static public String SHOW\_BUTTON\_CLASS\_NAME = "showBtn";

static public String BOOK\_TABLE\_TITLE = "Book table";

static public String ADD\_BOOK\_BTN = "addBookBtn";

static public String CLEAR\_BTN = "clearBtn";

private HtmlTemplater templater = new HtmlTemplater();

public MainServlet() throws Exception {

bookDatabase = new BookDatabase("dba", "goalie", "books");

log.info("MainServlet start");

}

public void init(ServletConfig config) throws ServletException {

System.out.println("Log4JInitServlet is initializing log4j");

String log4jLocation = config.getInitParameter("log4j-properties-location");

ServletContext sc = config.getServletContext();

if (log4jLocation == null) {

System.err.println("\*\*\* No log4j-properties-location init param, so initializing log4j with BasicConfigurator");

BasicConfigurator.configure();

} else {

String webAppPath = sc.getRealPath("/");

String log4jProp = webAppPath + log4jLocation;

File yoMamaYesThisSaysYoMama = new File(log4jProp);

if (yoMamaYesThisSaysYoMama.exists()) {

log.info("Initializing log4j with: " + log4jProp);

PropertyConfigurator.configure(log4jProp);

} else {

log.error("\*\*\* " + log4jProp + " file not found, so initializing log4j with BasicConfigurator");

BasicConfigurator.configure();

}

}

super.init(config);

}

public static SimpleDateFormat getDateFormat()

{

return new SimpleDateFormat("dd.MM.yyyy", Locale.ENGLISH);

}

public static String getStringDatePresentation(Date date)

{

return getDateFormat().format(date);

}

static public String getOutOfRangeMessage() {

return "Field \"Page amount\" must be unsigned number in range["

+ 0 + "; " + Integer.MAX\_VALUE + "]";

}

static public String getIncorrectDateMesage()

{

return "Field \"Publishing date\" must contain date";

}

static public String getEmptyMessage(String fieldName)

{

return "Field \"" + fieldName + "\" is empty!";

}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

if(request.getAttribute("showAddBookPage") == null) {

if(!(fieldIsEmpty(request, "name")

|| fieldIsEmpty(request, "author")

|| fieldIsEmpty(request, "pageAmount")

|| fieldIsEmpty(request, "publishingDate")

)

) {

addBook(request);

}

}

templater.setAttribute("errorMessage", "String", errorMessage);

printHtmlPage(response, templater.render("f:\\Study\\7\_Semester\\РИП\\Lab3\\WebContent\\AddBooks.html.template"));

}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

if(bookDatabase.isEmpty()) {

request.setAttribute("showAddBookPage", true);

doPost(request, response);

return;

}

List<Book> books;

try {

books = bookDatabase.getBooks();

templater.setAttribute("book", "Book", books.get(0));

printHtmlPage(response, templater.render("f:\\Study\\7\_Semester\\РИП\\Lab3\\WebContent\\BookTable.html.template"));

log.info("Open book table");

} catch (SQLException exception) {

String exceptionMessage = "SQLException, doGet bookDatabase.getBooks(): "

+ exception.getMessage();

log.error(exceptionMessage);

exception.printStackTrace();

}

}

private void printHtmlPage(HttpServletResponse response, String page) throws IOException {

response.setContentType("text/html");

java.io.PrintWriter out = response.getWriter();

out.print(page);

out.flush();

}

private void addBook(HttpServletRequest request)

{

String name = request.getParameter("name");

String author = request.getParameter("author");

if(!checkPublishingYear(request)) {

return;

}

String yearString = request.getParameter("pageAmount");

Integer pageAmount = Integer.parseUnsignedInt(yearString);

if(!checkPublishingData(request))

{

return;

}

String dateStr = request.getParameter("publishingDate");

DateFormat format = getDateFormat();

try {

java.sql.Date sqlDate = new java.sql.Date(format.parse(dateStr).getTime());

bookDatabase.insertName(new Book(name, author, pageAmount, sqlDate), 1);

errorMessage = new String();

log.info("Add book");

} catch (ParseException | SQLTimeoutException e) {

String exceptionMessage = "Publishing date incorrect. MainServlet.addBook()";

System.out.println(exceptionMessage);

log.error(exceptionMessage);

}

}

private boolean checkPublishingData(HttpServletRequest request) {

String dateStr = request.getParameter("publishingDate");

DateFormat format = getDateFormat();

try {

format.parse(dateStr);

} catch (ParseException e) {

errorMessage = getIncorrectDateMesage();

log.error(errorMessage);

return false;

}

return true;

}

private boolean checkPublishingYear(HttpServletRequest request)

{

try

{

String yearString = request.getParameter("pageAmount");

Integer.parseUnsignedInt(yearString);

} catch(NumberFormatException e) {

errorMessage = getOutOfRangeMessage();

log.error(errorMessage);

return false;

}

return true;

}

private boolean fieldIsEmpty(HttpServletRequest request, String fieldName)

{

if(request.getParameter(fieldName).isEmpty()) {

errorMessage = getEmptyMessage(fieldName);

log.error(errorMessage);

return true;

}

return false;

}

}

**MainServletTests.java**

**public** **class** MainServletTests {

**static** **private** String *PATH\_TO\_CHROME\_DRIVER* = "f:\\SDK\\GeckoDriver\\chromedriver.exe";

**static** **private** String *PATH\_TO\_MAIN\_SERVLET* = "http://localhost:8080/Lab3/MainServlet";

**static** **private** ChromeDriverService *service*;

**static** **private** WebDriver *driver*;

@BeforeClass

**public** **static** **void** createAndStartService() **throws** IOException{

*service* = **new** ChromeDriverService.Builder()

.usingDriverExecutable(**new** File(*PATH\_TO\_CHROME\_DRIVER*))

.usingAnyFreePort()

.build();

*service*.start();

*createDriver*();

}

@AfterClass

**public** **static** **void** createAndStopService() {

*driver*.quit();

*service*.stop();

}

**static** **public** **void** createDriver() {

*driver* = **new** RemoteWebDriver(*service*.getUrl(),

DesiredCapabilities.*chrome*());

*driver*.navigate().refresh();

*driver*.get(*PATH\_TO\_MAIN\_SERVLET*);

}

@Test

**public** **void** testEmptyField() {

testFields("", "a", "5", "12.12.2012", MainServlet.*getEmptyMessage*("name"));

testFields("n", "", "2", "12.12.2012", MainServlet.*getEmptyMessage*("author"));

testFields("c", "a", "", "12.12.2012", MainServlet.*getEmptyMessage*("pageAmount"));

testFields("c", "a", "3", "", MainServlet.*getEmptyMessage*("publishingDate"));

}

@Test

**public** **void** testPageAmountField() {

Calendar calendar = Calendar.*getInstance*();

calendar.set(2012, 12, 12);

Date rightDate = **new** Date(calendar.getTimeInMillis());

testFields(

"n",

"a",

"-9999999999",

"12.12.2012",

MainServlet.*getOutOfRangeMessage*()

);

testBook(

**new** Book("n", "a", Integer.***MIN\_VALUE***, rightDate),

MainServlet.*getOutOfRangeMessage*()

);

testBook(

**new** Book("n", "a", -1, rightDate),

MainServlet.*getOutOfRangeMessage*()

);

testFields(

"n",

"a",

"9999999999",

"12.12.2012",

MainServlet.*getOutOfRangeMessage*()

);

testBook(**new** Book("n", "a", 0, rightDate), **null**);

testBook(**new** Book("n", "a", 2017, rightDate), **null**);

testBook(**new** Book("n", "a", Integer.***MAX\_VALUE***, rightDate), **null**);

}

@Test

**public** **void** testPublishingDateFields() {

testFields(

"n",

"a",

"100",

"12.12.2012",

**null**

);

testFields(

"n",

"a",

"100",

"120.12.2012",

MainServlet.*getIncorrectDateMesage*()

);

testFields(

"n",

"a",

"100",

"2012",

MainServlet.*getIncorrectDateMesage*()

);

testFields(

"n",

"a",

"100",

"d",

MainServlet.*getIncorrectDateMesage*()

);

testFields(

"n",

"a",

"100",

"",

MainServlet.*getIncorrectDateMesage*()

);

}

@Test

**public** **void** testNameAndAuthorFields() {

testFields("a", "b", "1", "12.12.2012", **null**);

}

**private** **void** testBook(Book book, String servletMessage)

{

fillFields(book);

sumbmitData(servletMessage);

}

**private** **void** testFields(

String name,

String author,

String pageAmount,

String publishingDate,

String servletMessage

) {

fillFields(name, author, pageAmount, publishingDate);

sumbmitData(servletMessage);

}

**private** **void** sumbmitData(String servletMessage)

{

*driver*.findElement(By.*className*(MainServlet.*ADD\_BOOK\_BTN*)).click();

**if**(servletMessage != **null**) {

String errorMessage = *driver*.findElement(By.*tagName*("h1")).getText();

Assert.*assertEquals*(errorMessage, servletMessage);

clearFields();

}

}

@Before

**public** **void** clearFields()

{

*driver*.findElement(By.*className*("nameField")).clear();

*driver*.findElement(By.*className*("authorField")).clear();

*driver*.findElement(By.*className*("pageAmountField")).clear();

*driver*.findElement(By.*className*("publishingDateField")).sendKeys(Keys.***DELETE***);

}

**private** **void** fillFields(Book book)

{

*driver*.findElement(By.*className*("nameField")).sendKeys(book.name);

*driver*.findElement(By.*className*("authorField")).sendKeys(book.author);

String integerStr = Integer.*toString*(book.pageAmount);

*driver*.findElement(By.*className*("pageAmountField")).sendKeys(integerStr);

String dateStr = MainServlet.*getDateFormat*().format(book.publishingData);

*driver*.findElement(By.*className*("publishingDateField")).sendKeys(dateStr);

}

**private** **void** fillFields(

String name,

String author,

String pageAmount,

String publishingDate

) {

*driver*.findElement(By.*className*("nameField")).sendKeys(name);

*driver*.findElement(By.*className*("authorField")).sendKeys(author);

*driver*.findElement(By.*className*("pageAmountField")).sendKeys(pageAmount);

*driver*.findElement(By.*className*("publishingDateField")).sendKeys(publishingDate);

}

}

**VariableExtractTests.java**

**class** VariableExtractTests {

**private** HtmlTemplater templater = **new** HtmlTemplater();

@Test

**void** checkExtractTemplateVariables() {

List<String> emptyList = **new** ArrayList<>();

testExtractTemplateVariables("a", emptyList);

testExtractTemplateVariables("aaa", emptyList);

testExtractTemplateVariables("12", emptyList);

testExtractTemplateVariables("-12", emptyList);

testExtractTemplateVariables("-0", emptyList);

testExtractTemplateVariables("0", emptyList);

testExtractTemplateVariables("name", emptyList);

testExtractTemplateVariables("book.name", emptyList);

testExtractTemplateVariables("$name", emptyList);

testExtractTemplateVariables("$book.name", emptyList);

testExtractTemplateVariables("$", emptyList);

testExtractTemplateVariables("$(", emptyList);

testExtractTemplateVariables("$()", emptyList);

List<String> resultList = **new** ArrayList<>();

resultList.add("$(name)");

testExtractTemplateVariables("$(name)", resultList);

resultList.clear();

resultList.add("$(book.name)");

testExtractTemplateVariables("$(book.name)", resultList);

resultList.clear();

resultList.add("$(book.name)");

resultList.add("$(var)");

testExtractTemplateVariables("<t>$(book.name)</t>\n$(var)", resultList);

}

@Test

**void** checkExtractVariables() {

List<String> resultList = **new** ArrayList<>();

resultList.add("name");

testExtractVariables("$(name)", resultList);

resultList.clear();

resultList.add("book");

resultList.add("name");

testExtractVariables("$(book.name)", resultList);

}

@Test

**void** checkGetValueForSimpleType() {

Calendar calendar = Calendar.*getInstance*();

calendar.set(2012, 12, 12);

Date date = **new** Date(calendar.getTimeInMillis());

templater.setAttribute("strVar", "String", "theString");

templater.setAttribute("intVar", "Integer", **new** Integer(4));

templater.setAttribute("dateVar", "Date", date);

testGetValueForSimpleTemplate("strVar", "theString");

testGetValueForSimpleTemplate("intVar", "4");

//testGetValueForSimpleTemplate("dateVar", "2012-12-12");// **TODO** : might incorrect convert to string

}

@Test

**void** checkGetValueForBook() {

Calendar calendar = Calendar.*getInstance*();

calendar.set(2012, 12, 12);

Date date = **new** Date(calendar.getTimeInMillis());

Book testBook = **new** Book("n", "a", 9, date);

templater.setAttribute("bookVar", "Book", testBook);

List<String> resultList = **new** ArrayList<>();

resultList.add("bookVar");

testGetValue(resultList, "");

resultList.add("name");

testGetValue(resultList, testBook.name);

resultList.remove(1);

resultList.add("author");

testGetValue(resultList, testBook.author);

resultList.remove(1);

resultList.add("pageAmount");

testGetValue(resultList, testBook.pageAmount.toString());

resultList.remove(1);

resultList.add("publishingData");

testGetValue(resultList, testBook.publishingData.toString());

}

@Test

**void** checkConvertString() {

templater.setAttribute("strVar", "String", "theString");

Calendar calendar = Calendar.*getInstance*();

calendar.set(2012, 12, 12);

Date date = **new** Date(calendar.getTimeInMillis());

Book testBook = **new** Book("n", "a", 9, date);

templater.setAttribute("book", "Book", testBook);

testConvertString(

"<t>$(book.name)</t>\n$(strVar)",

"<t>n</t>\ntheString"

);

testConvertString(

"<tr>"

+ "<td>$(book.name)</td>"

+ "<td>$(book.author) </td>"

+ "<td>$(book.pageAmount)</td>"

+ "<td>$(book.publishingData)</td>"

+ "</tr>",

"<tr>"

+ "<td>n</td>"

+ "<td>a </td>"

+ "<td>9</td>"

+ "<td>2013-01-12</td>"

+ "</tr>"

);

}

**private** **void** testConvertString(String string, String convertString)

{

String result = templater.convertToTemplateString(string);

Assert.*assertEquals*(result, convertString);

}

**private** **void** testGetValue(List<String> variableParts, String rightResult) {

String result = templater.getValue(variableParts);

Assert.*assertEquals*(result, rightResult);

}

**private** **void** testGetValueForSimpleTemplate(String name, String value)

{

List<String> resultList = **new** ArrayList<>();

resultList.add(name);

testGetValue(resultList, value);

}

**public** **void** testExtractVariables(String string, List<String> rightParts)

{

List<String> result = HtmlTemplater.*extractVariablePart*(string);

Assert.*assertEquals*(result, rightParts);

}

**public** **void** testExtractTemplateVariables(String string, List<String> rightResult)

{

List<String> result = HtmlTemplater.*extractTemplateVariables*(string);

Assert.*assertEquals*(result, rightResult);

}

}

**HtmlTemplater.java**

**public** **class** HtmlTemplater {

**private** **static** String *VARIABLE\_TEMPLATE* = "[\\w]+";

**private** **static** String *VARIABLE\_AND\_FIELD* = *VARIABLE\_TEMPLATE* +

"(." + *VARIABLE\_TEMPLATE* + ")\*";

**private** **static** Pattern *variablePattern* = Pattern.*compile*(*VARIABLE\_TEMPLATE*);

**private** Map<String, TemplateVariable> variables = **new** HashMap<String, TemplateVariable>();

**public** String render(String fileName)

{

String page = "";

BufferedReader br = **null**;

FileReader fr = **null**;

**try** {

//br = new BufferedReader(new FileReader(FILENAME));

fr = **new** FileReader(fileName);

br = **new** BufferedReader(fr);

String sCurrentLine;

**while** ((sCurrentLine = br.readLine()) != **null**) {

page += convertToTemplateString(sCurrentLine);

}

**return** page;

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

**if** (br != **null**) {

br.close();

}

**if** (fr != **null**) {

fr.close();

}

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

**return** page;

}

// The method is public because need to test it

**static** **public** List<String> extractTemplateVariables(String string)

{

List<String> variables = **new** ArrayList<>();

Pattern p = Pattern.*compile*("\\$\\(" + *VARIABLE\_AND\_FIELD* +"\\)");

Matcher matcher = p.matcher(string);

**while**(matcher.find()) {

variables.add(string.substring(matcher.start(), matcher.end()));

}

**return** variables;

}

**static** **public** List<List<String>> extractVariables(List<String> templates)

{

List<List<String>> variables = **new** ArrayList<>();

**for**(**int** i = 0; i < templates.size(); i++)

{

List<String> callPart = *extractVariablePart*(templates.get(i));

variables.add(callPart);

}

**return** variables;

}

**static** **public** List<String> extractVariablePart(String string)

{

Matcher matcher = *variablePattern*.matcher(string);

List<String> callPart = **new** ArrayList<>();

**while**(matcher.find()) {

callPart.add(

string.substring(matcher.start(), matcher.end())

);

}

**return** callPart;

}

// **TODO** : do private

**public** String convertToTemplateString(String string)

{

List<String> templates = *extractTemplateVariables*(string);

List<List<String>> variableParts = *extractVariables*(templates);

List<String> values = **new** ArrayList<String>();

**for**(**int** i = 0; i < variableParts.size(); i++) {

String value = getValue(variableParts.get(i));

values.add(value);

}

String convertString = string;

**for**(**int** i = 0; i < templates.size(); i++) {

convertString = convertString.replace(templates.get(i), values.get(i));

}

**return** convertString;

}

// The method is public because need to test it

**public** String getValue(List<String> variableParts)

{

TemplateVariable variable = variables.get(variableParts.get(0));

**if**(variable != **null**) {

**return** extractValue(variable, variableParts);

}

**return** "";

}

**public** **void** setAttribute(String name, String type, Object object)

{

variables.put(name, **new** TemplateVariable(type, object));

}

**private** String extractValue(TemplateVariable variable, List<String> variableParts)

{

**switch** (variable.type) {

**case** "String":

**return** variable.object.toString();

**case** "Integer":

**return** ((Integer) variable.object).toString();

**case** "Date":

**return** ((Date) variable.object).toString();

**case** "Book":

**return** extractValueToBook(variable, variableParts);

**default**:

**break**;

}

**return** "";

}

**private** String extractValueToBook(TemplateVariable variable, List<String> variableParts)

{

**if**(variableParts.size() > 1) {

**switch** (variableParts.get(1)) {

**case** "name":

**return** ((Book) variable.object).name;

**case** "author":

**return** ((Book) variable.object).author;

**case** "pageAmount":

**return** ((Book) variable.object).pageAmount.toString();

**case** "publishingData":

**return** ((Book) variable.object).publishingData.toString();

**default**:

**break**;

}

}

**return** "";

}

}

**AddBooks.html.template**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Books</title>

</head>

<body>

<h1>$(errorMessage)</h1>

<form action = "MainServlet" method = "POST">

Book Name: <input type = "text" name = "name"

class="nameField">

<br/>

Author: <input type = "text" name = "author"

class="authorField">

<br/>

Page amount: <input type = "number" name = "pageAmount"

class="pageAmountField">

<br/>

Publishing date: <input type = "date" name = "publishingDate"

class="publishingDateField">

<br/>

<input type = "submit" value = "Add book" class="addBookBtn" />

</form>

<form action = "MainServlet" method="GET">

<input type = "submit" value = "Show my books"

class="showBtn"/>

</form>

</body>

</html>

**BookTable.html.template**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Book table</title>

</head>

<body>

<table>

<tr>

<th>Name</th>

<th>Author</th>

<th>Page amount</th>

<th>Publishing date</th>

</tr>

<tr>

<td>$(book.name)</td>

<td>$(book.author) </td>

<td>$(book.pageAmount)</td>

<td>$(book.publishingData)</td>

</tr>

</table>

</body>

</html>

# Использованные инструменты

1. Язык Java
2. База данных NuoDB
3. Сервер Tomcat