Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Инженерно-экономический факультет

Кафедра экономической информатики

Н. О. Петрович, О. Голда, Д. А. Сторожев

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ РАЗРАБОТКИ СЕРВЕРНОЙ ЧАСТИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

*Рекомендовано УМО по образованию в области информатики
и радиоэлектроники
в качестве учебно-методического пособия
для специальности 6-05-0611-01 «Информационные системы и технологии»*

Минск БГУИР 2026

УДК 004.42(075.8)
ББК 32.973.3я73
П30

Р е ц е н з е н т ы:

кафедра информатики и веб-дизайна
учреждения образования «Белорусский государственный
технологический университет»
(протокол № 3 от 25.10.2024);

кафедра программного обеспечения информационных технологий
учреждения образования «Белорусский государственный университет
информатики и радиоэлектроники»
(протокол № 11 от 02.03.2025);

кафедра автоматизированных систем управления производством
учреждения образования «Белорусский государственный аграрный
технический университет»
(протокол № 4 от 25.10.2024);

ведущий инженер-программист
ООО «Финмаркет Софт»
А. С. Сушинский

Петрович, Н. О.

П30 Современные технологии разработки серверной части программных приложений : учеб.-метод. пособие / Н. О. Петрович, О. Голда, Д. А. Сторожев. – Минск : БГУИР, 2026. – 176 с. : ил.

ISBN 978-985-543-852-7.

Рассмотрены назначения, функции и структура технологий реализации гибкого и функционального программного обеспечения, принципы разработки и проектирования архитектуры систем на основе современных подходов и тенденций развития технологий с применением фреймворков Hibernate и Spring Framework, изложен механизм реализации персистентности. Представлены темы, посвященные проектированию и реализации архитектуры на основе стиля REST и принципов SOLID.

Предназначено для студентов, обучающихся по специальности 6-05-0611-01 «Информационные системы и технологии».

УДК 004.42(075.8)
ББК 32.973.3я73

ISBN 978-985-543-852-7

Содержание

Тема 1. Сборка проекта на основе *Maven*.....5
1.1 Установка пакета *Maven*.....5
1.2 Проверка поддержки *Maven*.....9
1.3 Интеграция *Maven* в среду разработки *Idea*.....9
1.4 Управление жизненным циклом проекта.....12
1.5 Создание профилей сборки проектов.....15
1.6 Управление репозиториями *Maven*.....17
1.7 Интеграция плагинов для создания проекта.....19
1.8 Создание и сборка проекта с интеграцией зависимостей.....20
Тема 2. Системы контроля версий на примере *Git*.....22
2.1 Настройка рабочего пространства и регистрация в системе.....22
2.2 Установка дистрибутива *Git*.....25
2.3 Настройка конфигурации системы контроля версии.....25
2.4 Создание и инициализация репозиториев.....27
2.5 Просмотр истории версий.....30
2.6 Клонирование существующего репозитория.....32
2.7 Работа с удаленными репозиториями.....33
Тема 3. Управление персистенцией на основе *JPA*.....35
3.1 Проектирование структуры проекта.....36
3.2 Структура БД.....61
3.3 Создание *Entity*-классов для маппинга объектной и реляционной модели.....62
Тема 4. Реализация веб-приложения на основе *Servlets*.....69
4.1 Конфигурация проекта.....69
4.2 Жизненный цикл *Servlets*.....72
4.3 Добавление *JSP*.....81
4.4 Передача параметров *HTTP*-запроса.....94
4.5 Добавление форм фронт-составляющей.....98
4.6 Внедрение проверок.....99
4.7 Работа с возвратом списка.....102
4.8 Добавление *JSTL*.....105
4.9 Реализация перенаправления между сервлетами.....106
4.10 Редактирование списка.....108
4.11 Добавление *JQuery* и *Bootstrap*.....111
4.12 Реализация фильтра.....120
4.13 Конфигурация БД.....124

4.14 Работа с регистрацией 133
4.15 Хеширование паролей 140
4.16 Добавление поддержки *Cookie* 148
Тема 5. Основы *Spring Framework* 157
5.1 Конфигурация проекта 157
5.2 Основы *Spring Core*. 162
5.3 Итеграция *Spring Boot* 170
Список использованных источников 175

Тема 1. Сборка проекта на основе *Maven*

Maven – это инструмент для автоматической сборки проектов на *Java* и других языках программирования. Он помогает разработчикам правильно подключить библиотеки и фреймворки, управлять их версиями, выстроить структуру проекта и составить к нему документацию.

Например, чтобы собрать приложение для управления базами данных на *Java*, нам понадобятся фреймворки *Spring* и *Hibernate*, библиотека *JUnit* для модульного тестирования и сама база данных. Все это можно собрать в одном проекте вручную, но могут быть трудности из-за большого числа зависимостей. Здесь на помощь разработчикам приходит *Maven*. Он автоматически добавит эти или другие зависимости в проект и соберет его в исполняемый файл.

Maven – не единственный сборщик проектов. Некоторые разработчики используют его аналоги – *Gradle* и *Ant*, но именно *Maven* сегодня – золотой стандарт в индустрии.

1.1 Установка пакета *Maven*

Для работы необходимо установить *Maven* на компьютер, соответственно переходим на официальный сайт *Maven* и скачиваем *Binary zip archive*, но перед этим убедитесь, что у вас на компьютере установлен *JDK* (рисунок 1.1).

Открываем командную строку и пишем команду:

java -version

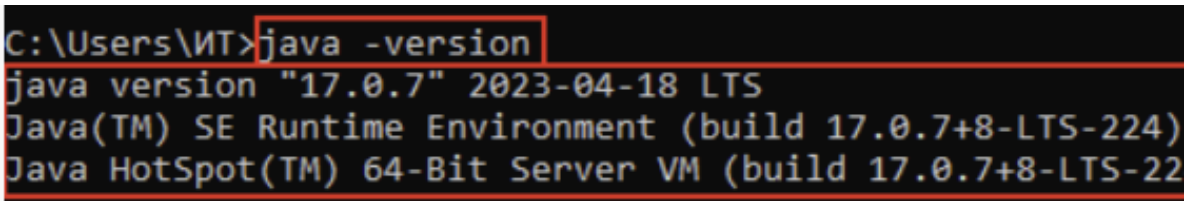


Рисунок 1.1 – Проверка версии *Java*

Установите *Maven*. Настройте *environment variables* и проверьте, что *Maven* настроен корректно.

Скачайте архив *binary* по ссылке *http://maven.apache.org/download.cgi*. Архив находится в разделе *Files* (рисунок 1.2).

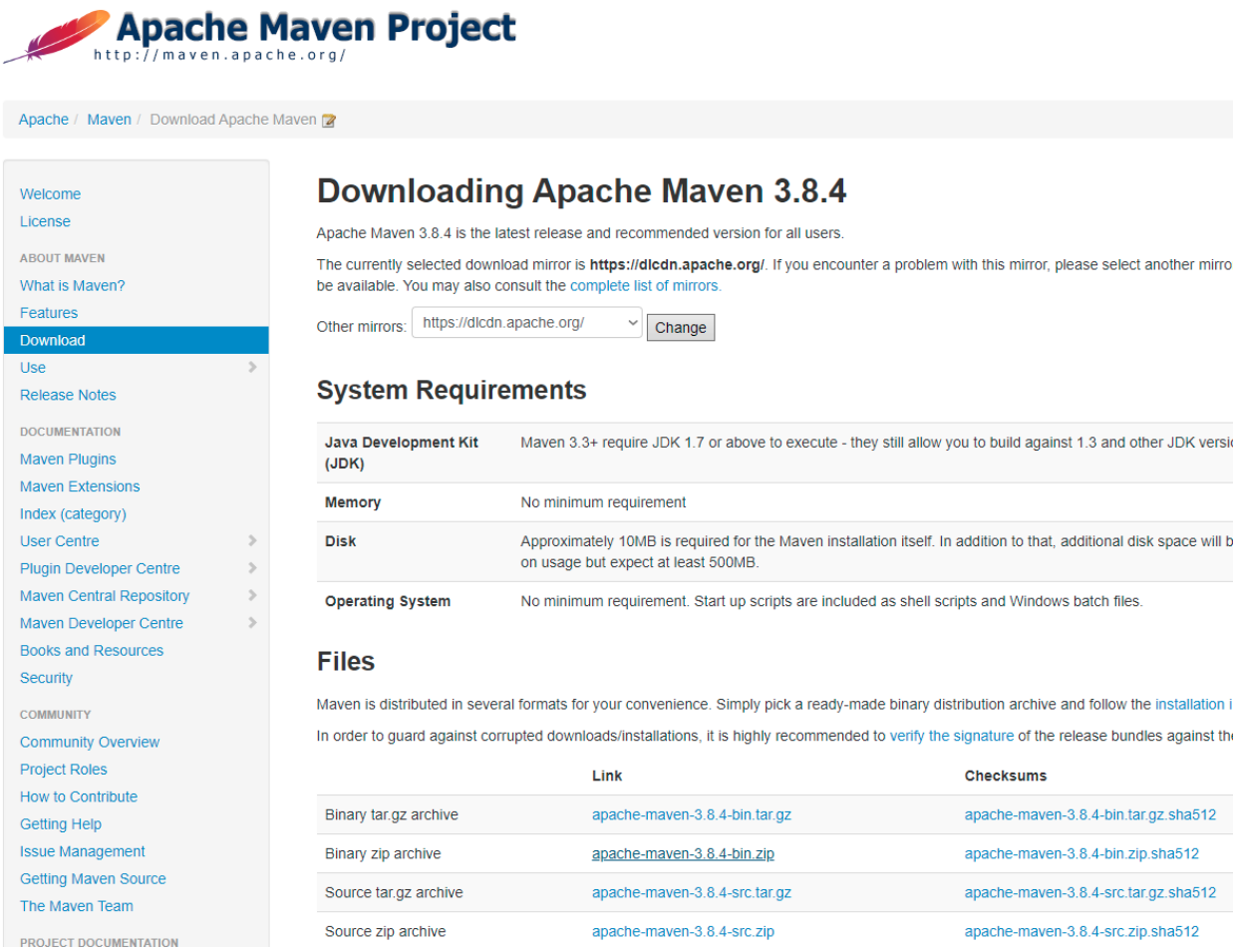


Рисунок 1.2 – Архив сборщика *Maven*

Обратите внимание, что понадобится версия *JDK* 1.7 и выше. После загрузки вы получите *zip*-архив около 10 Мб.

Maven устанавливается просто копированием в нужную директорию – никакого инсталлера нет. Распакуйте архив в любую папку. Для *Windows*, как правило, путь к папке не должен содержать пробелов.

Для начала использования *Maven* необходимо настроить переменные среды *Windows*. Переменная *MAVEN_HOME (M2_HOME)* должна быть установлена, а переменная *PATH* должна быть модифицирована для включения папки, откуда запускается *Maven* (рисунок 1.3).

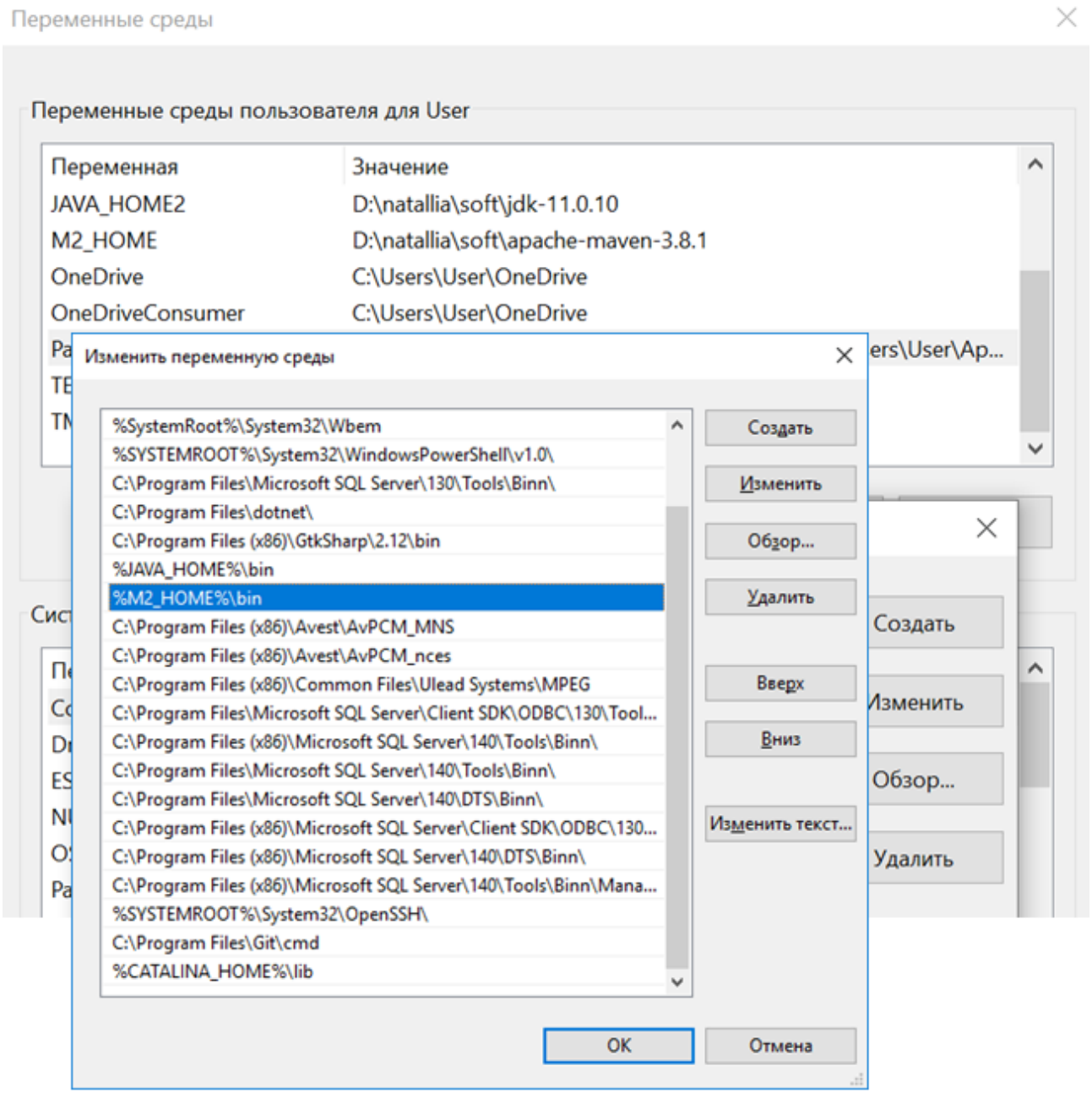


Рисунок 1.3 – Управление переменными средами для установки *Maven*

Убедитесь, что установлены следующие компоненты, как указано на рисунках 1.4 и 1.5.

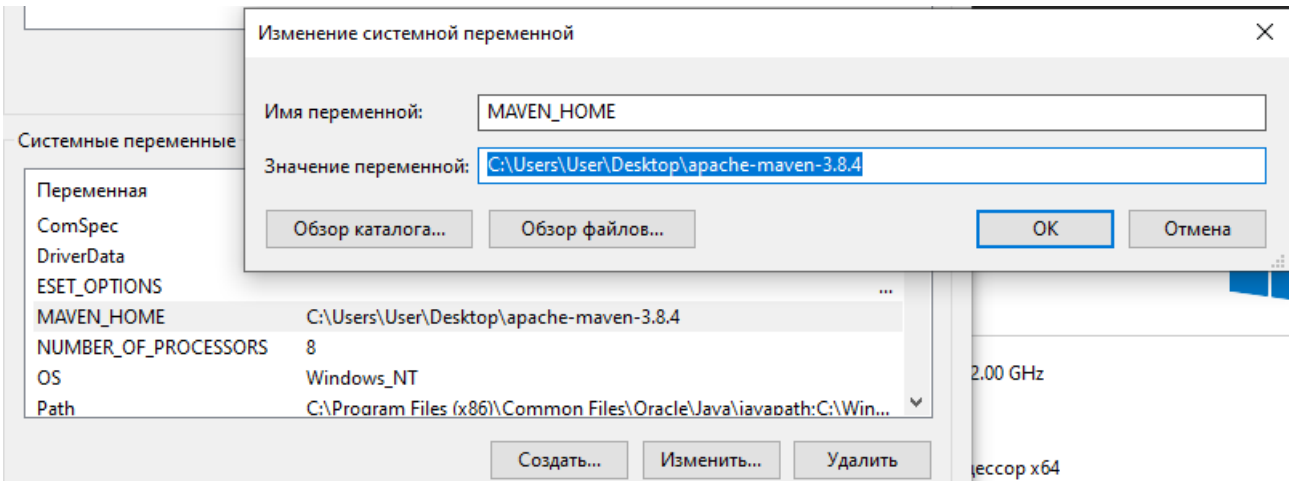


Рисунок 1.4 – Изменение системной переменной

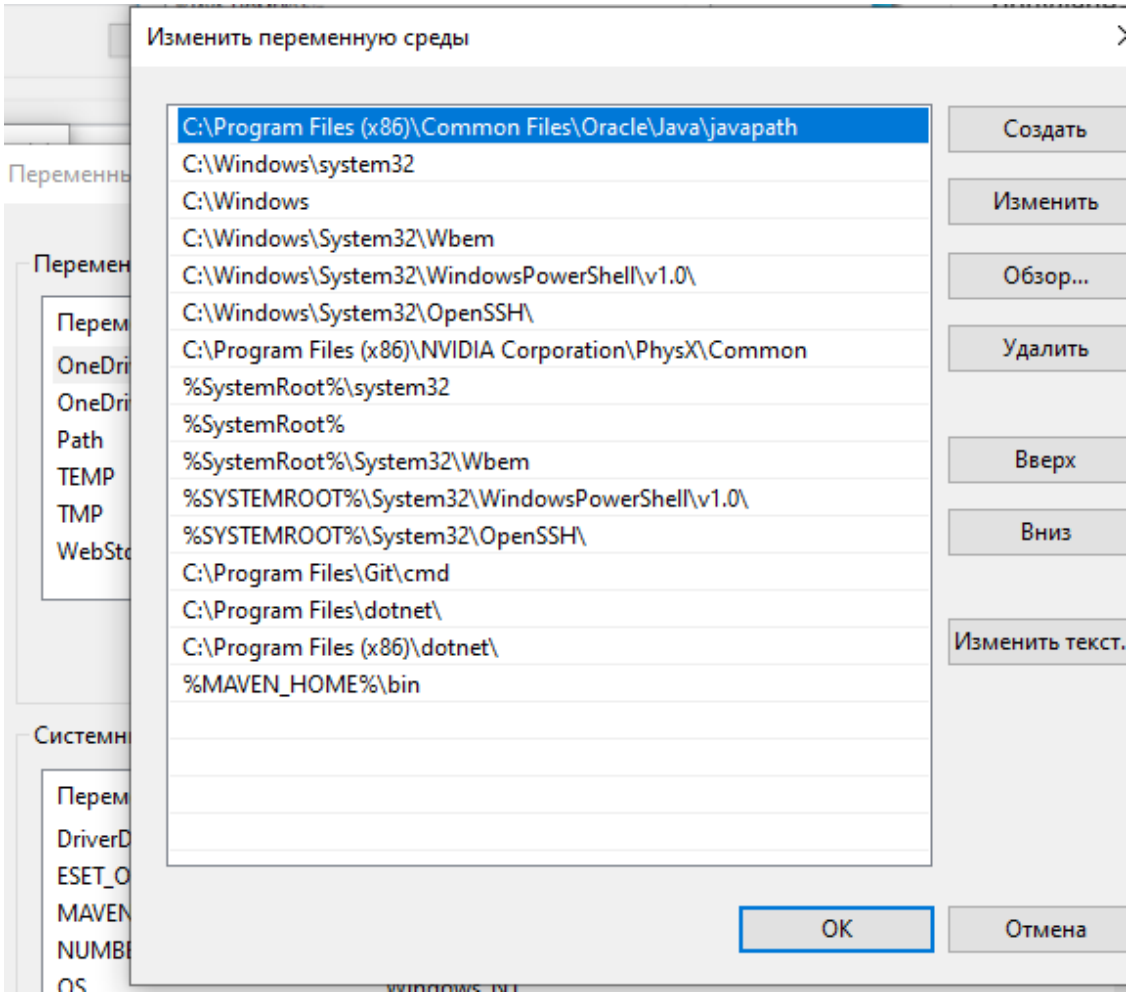


Рисунок 1.5 – Изменение переменной среды

Apache Maven теперь готов к использованию. Он также доступен для интеграции с *IDE* и другими программными средствами, предназначенными для разработки.

1.2 Проверка поддержки *Maven*

Для проверки можно запустить командную строку и выполнить команду *mvn-version* (рисунок 1.6).

```
:\Users\User>mvn -version
pache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
aven home: C:\Users\User\Desktop\apache-maven-3.8.4
ava version: 10.0.2, vendor: Oracle Corporation, runtime: C:
efault locale: ru_RU, platform encoding: Cp1251
S name: "windows 10", version: "10.0", arch: "amd64", family
```

Рисунок 1.6 – Проверка работы *Maven* в консоли *cmd*

Шаблон, на основе которого будут создаваться проекты, включает поддержку *Maven*. Если нужно будет поменять директорию *Maven* или репозиторий, то необходимо зайти в среду разработки, в данном примере *Idea*. Затем перейти в настройки *Maven home directory* (рисунок 1.7).

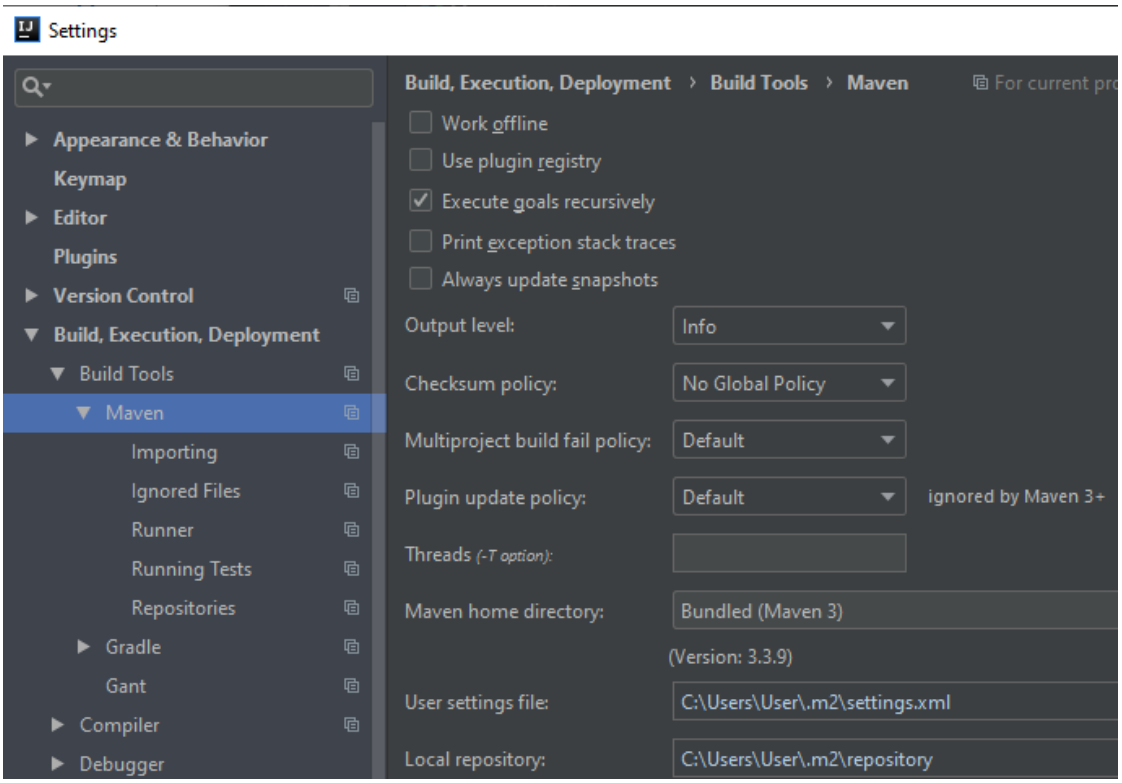


Рисунок 1.7 – Проверка настройки *Maven home directory* в среде *Idea*

1.3 Интеграция *Maven* в среду разработки *Idea*

Создайте проект с помощью *Maven*. Для этого перейдите по следующим разделам среды разработки (рисунки 1.8 и 1.9).

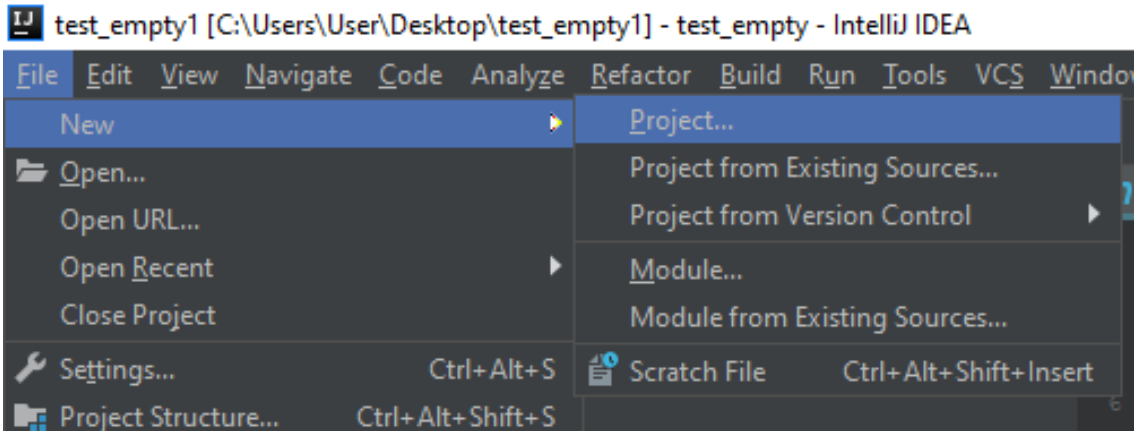


Рисунок 1.8 – Создание проекта в среде *Idea*

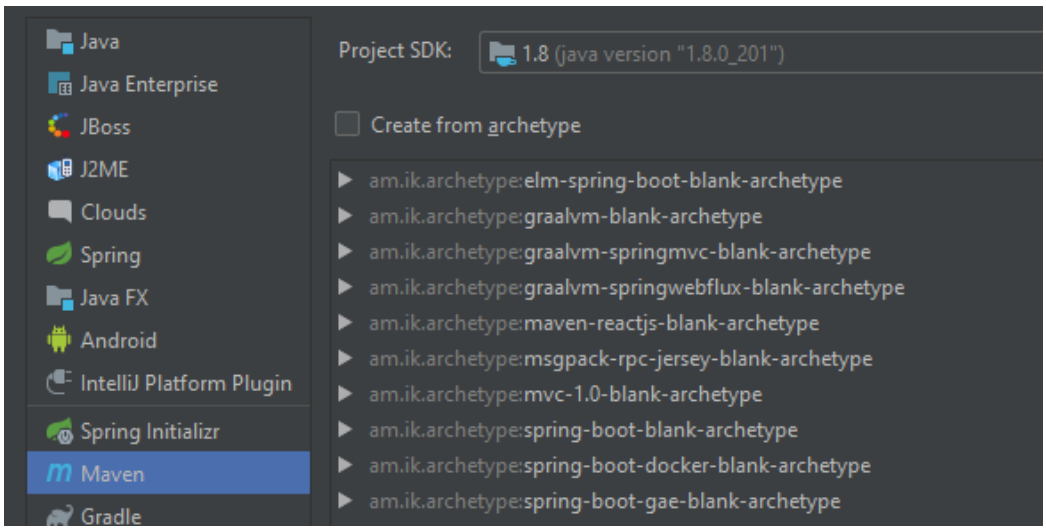


Рисунок 1.9 – Выбор сборщика проектов и архетипа

Выбираем *Maven*. Архетипы будут рассмотрены в следующем подразделе. Задаем имя проекта, а также спецификацию. После чего должен быть автоматически создан проект с *pom.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>test_empty</groupId>
    <artifactId>test_empty</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <name>Servlet</name>
    <packaging>war</packaging>
    <properties>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.source>1.8</maven.compiler.source>
        <junit.version>5.7.0</junit.version>
    </properties>
</project>
```

Изучите раздел с существующими зависимостями. У вас должно получиться примерно так после добавления.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <execution>
                    <id>id.clean</id>
                    <phase>clean</phase>
                    <goals>
                        <goal>run</goal>
                    </goals>
                    <configuration>
                        <tasks>
                            <echo>clean phase</echo>
                        </tasks>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Далее можно увидеть добавление зависимостей в проект (рисунок 1.10).

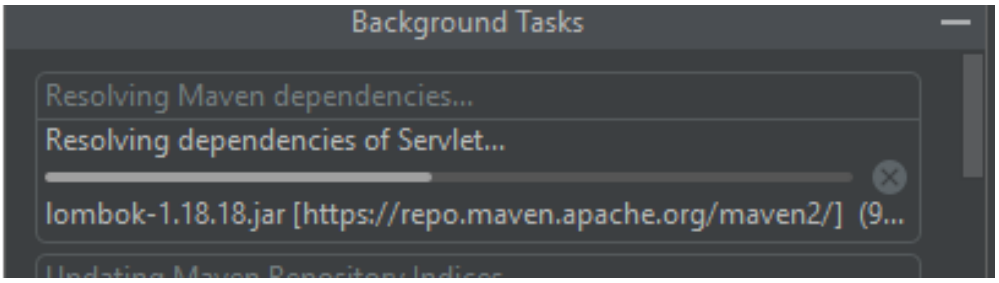


Рисунок 1.10 – Загрузка зависимостей с удаленного репозитория

После чего выполните команду, как показано на рисунке 1.11.

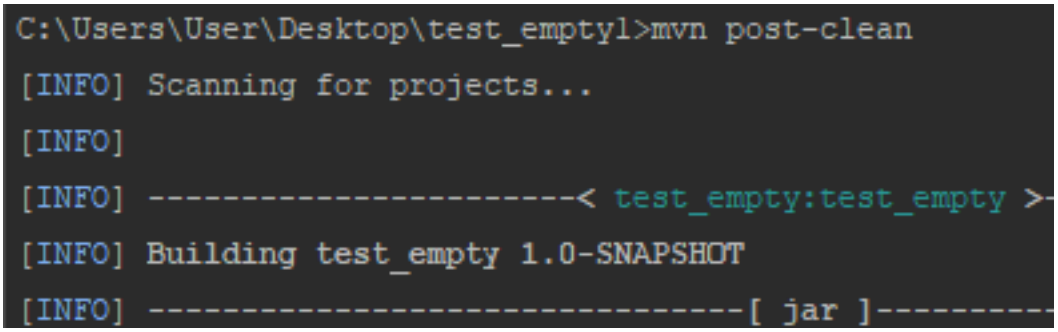


Рисунок 1.11 – Сборка и отчистка проекта

Maven начнет обработку и отображение всех фаз чистого жизненного цикла.

1.4 Управление жизненным циклом проекта

Это основной жизненный цикл *Maven*, который используется для создания приложения. Он имеет следующую 21 фазу (таблица 1.1).

Таблица 1.1 – Основные фазы жизненного цикла *Maven*

Название фазы	Описание
validate	Подтверждает, является ли проект корректным и вся ли необходимая информация доступа для завершения процесса сборки
initialize	Инициализирует состояние сборки, например, различные настройки
generate-sources	Включает любой исходный код в фазу компиляции
process-sources	Обрабатывает исходный код (подготавливает). Например, фильтрует определенные значения
generate-resources	Генерирует ресурсы, которые должны быть включены в пакет
process-resources	Копирует и отправляет ресурсы в указанную директорию. Это фаза перед упаковкой

Название фазы	Описание
compile	Компилирует исходный код проекта
process-classes	Обрабатывает файлы, полученные в результате компиляции. Например, оптимизация байт-кода *Java*-классов
generate-test-sources	Генерирует любые тестовые ресурсы, которые должны быть включены в фазу компиляции
process-test-sources	Обрабатывает исходный код тестов. Например, фильтрует значения
test-compile	Компилирует исходный код тестов
process-test-classes	Обрабатывает файлы, полученные в результате компиляции исходного кода тестов
test	Запускает тесты, используя приемлемый фреймворк юнит-тестирования (например, *Junit*)
prepare-package	Выполняет все необходимые операции для подготовки пакета непосредственно перед упаковкой
package	Преобразует скомпилированный код и пакет в дистрибутивный формат. Такие, как *JAR, WAR* или *EAR*
pre-integration-test	Выполняет необходимые действия перед выполнением интеграционных тестов
integration-test	Обрабатывает и распаковывает пакет, если необходимо, в среду, где будут выполняться интеграционные тесты
post-integration-test	Выполняет действия, необходимые после выполнения интеграционных тестов. Например, освобождение ресурсов
verify	Выполняет любые проверки для подтверждения того, что пакет пригоден и отвечает критериям качества
install	Устанавливает пакет в локальный репозиторий, который может быть использован как зависимость в других локальных проектах
deploy	Копирует финальный пакет (архив) в удаленный репозиторий для того, чтобы сделать его доступным другим разработчикам и проектам

Теперь обновим *pom*-файл и разберем, в чем он нам поможет сейчас.

Когда мы выполняем команду *Maven*, например *install*, то будут выполнены фазы до *install* и фаза *install*.

Различные задачи *Maven* будут привязаны к различным фазам жизненного цикла *Maven* в зависимости от типа архива (*JAR/WAR/EAR*).

В следующем примере мы привязываем задачу *maven-antrun-plugin:run* к нескольким фазам жизненного цикла сборки. Это также позволяет нам вызывать текстовые сообщения, отображая фазу жизненного цикла.

Содержание *pom*-файла должно выглядеть следующим образом:

```
  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <executions>
                <execution>
                    <id>id.compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>run</goal>
                    </goals>
                    <configuration>
                        <tasks>
                            <echo>compile phase</echo>
                        </tasks>
                    </configuration>
                </execution>
                <execution>
                    <id>id.test</id>
                    <phase>test</phase>
                    <goals>
                        <goal>run</goal>
                    </goals>
                    <configuration>
                        <tasks>
                            <echo>test phase</echo>
                        </tasks>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Далее запускаем сборку проекта, как показано на рисунке 1.12.

```
C:\Users\User\Desktop\test_empty1>mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< test_empty:test_empty >-
[INFO] Building test_empty 1.0-SNAPSHOT
```

Рисунок 1.12 – Начало сборки проекта

1.5 Создание профилей сборки проектов

Профиль сборки – это набор значений конфигурации, которые можно использовать для установки или переопределения значений по умолчанию сборки *Maven*. Используя профиль сборки, вы можете настроить сборку для различных сред.

Создадим структуру проекта, как показано на рисунке 1.13, для реализации тестовых профилей сборки проекта.

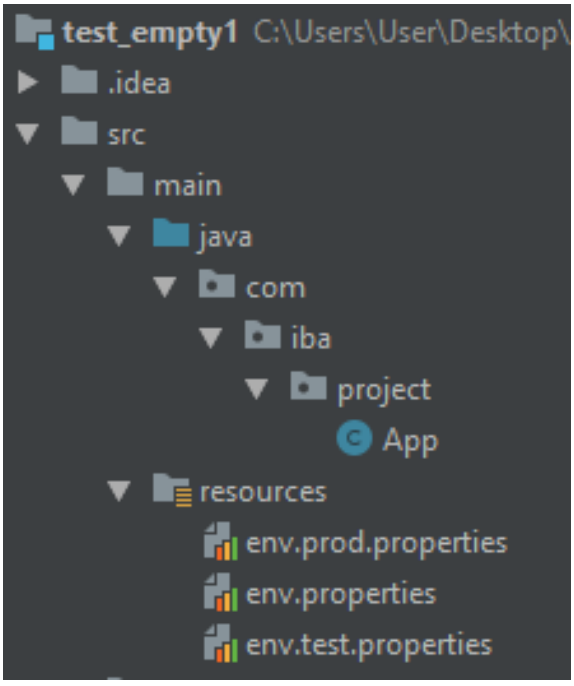


Рисунок 1.13 – Структура проекта с тестовыми профилями сборки

Теперь в каталоге *src/main/resources* есть три специальных файла для среды:

– *env.properties* – используется конфигурация по умолчанию, если профиль не указан;

– *env.test.properties* – тестовая конфигурация при использовании тестового профиля;

– *env.prod.properties* – производственная конфигурация при использовании профиля *prod*.

Явная активация профиля.

В следующем примере мы добавим *maven-antrun-plugin*: запустите цель, чтобы проверить фазу. Это позволит нам отображать текстовые сообщения для разных профилей. Мы будем использовать *pom.xml* для определения различных профилей и активировать профиль в командной консоли с помощью команды *Maven*.

```
<profiles>
    <profile>
        <id>test</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-antrun-plugin</artifactId>
                    <version>1.1</version>
                    <executions>
                        <execution>
                            <phase>test</phase>
                            <goals>
                                <goal>run</goal>
                            </goals>
                            <configuration>
                                <tasks>
                                    <echo>Using env.test.properties</echo>
                                    <copy
file="src/main/resources/env.test.properties"
tofile="${project.build.outputDirectory}/env.properties"/>
                                </tasks>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

В файл *properties* добавьте текстовую информацию.

В результате в сборку проекта будут включены данные настройки, указанные в тестовом файле или файле продакшина.

После чего выполните команду, как показано на рисунке 1.14.

```
C:\Users\User\Desktop\test_empty1>mvn test -Ptest
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< test_empty:test_empty >--
[INFO] Building test_empty 1.0-SNAPSHOT
```

Рисунок 1.14 – Выполнение тестового профиля сборки

1.6 Управление репозиториями *Maven*

При работе с *Maven* под репозиторием мы понимаем директорию, где хранятся все *JAR*, библиотеки, плагины и любые артефакты, которыми *Maven* может воспользоваться.

Существует три типа репозиториев *Maven*:

– локальные (*local*);

– центральные (*central*);

– удаленные (*remote*).

Пример использования удаленного репозитория начинается с поиска необходимой библиотеки, используя ресурс удаленного репозитория *https://mvnrepository.com* (рисунки 1.15 и 1.16).

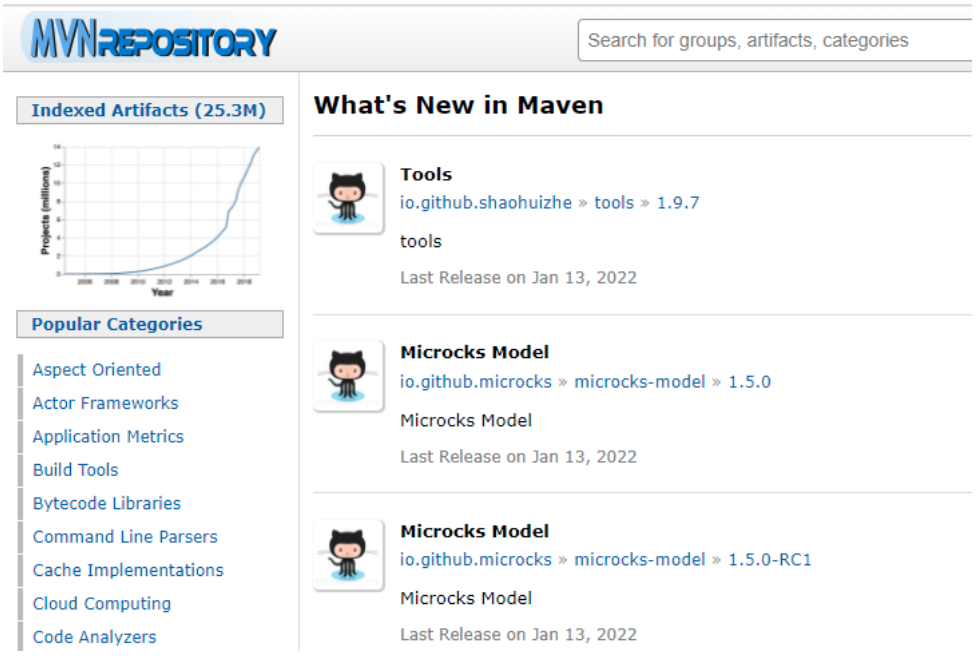


Рисунок 1.15 – Поиск необходимой библиотеки в удаленном репозитории

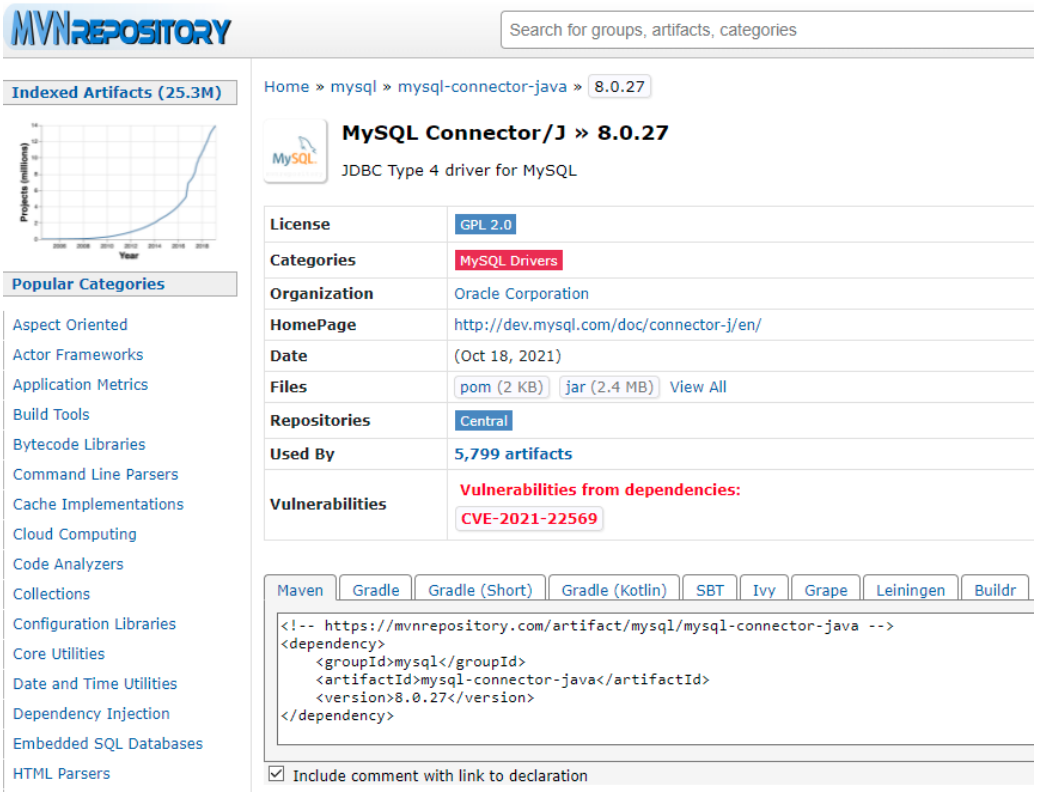


Рисунок 1.16 – Структура представления библиотеки в удаленном репозитории

Можем скопировать и использовать в своем проекте. Все библиотеки будут доступны нам в коде автоматически.

Последовательность поиска зависимостей *Maven*.

Когда мы выполняем команды сборки *Maven*, *Maven* начинает поиск библиотек зависимостей в следующей последовательности:

Шаг 1. Поиск зависимости в локальном репозитории. Если не найдена, перейти к шагу 2, иначе выполнить дальнейшую обработку.

Шаг 2. Поиск зависимости в центральном репозитории. Если не найдена и упоминается (упоминается удаленный репозиторий), перейдите к шагу 4. В противном случае он загружается в локальный репозиторий для дальнейшего использования.

Шаг 3. Если удаленный репозиторий не был упомянут, *Maven* просто останавливает обработку и выдает ошибку «Невозможно найти зависимость».

Шаг 4. Поиск зависимости в удаленном репозитории или репозиториях. Если она найдена, он загружается в локальный репозиторий для дальнейшего использования. В противном случае *Maven* останавливает обработку и выдает ошибку «Невозможно найти зависимость».

1.7 Интеграция плагинов для создания проекта

Если говорить в целом, то *Maven* – это фреймворк, который выполняет плагины. В этом фреймворке каждая задача, по сути, выполняется с помощью плагинов.

Плагины *Maven* используются для:

– создания *jar*-файла;

– создания *war*-файла;

– компиляции кода файлов;

– юнит-тестирования кода;

– создания отчетов проекта;

– создания документации проекта.

Типы плагинов

Существует два типа плагинов в *Maven*:

1) плагины сборки. Выполняются в процессе сборки и должны быть конфигурированы внутри блока <*build*></*build*> файла *pom.xml;*

2) плагины отчетов. Выполняются в процесса генерирования сайта и должны быть конфигурированы внутри блока <*reporting*></*reporting*> файла *pom.xml*.

Ниже приведен список наиболее используемых плагинов:

– *clean* – очищает цель после сборки. Удаляет директорию *target*;

– *compiler* – компилирует исходные *Java*-файлы;

– *surefire* – запускает тесты *JUnit*. Создает отчеты о тестировании;

– *jar* – собирает *JAR*-файл текущего проекта;

– *war* – собирает *WAR*-файл текущего проекта;

– *javadoc* – генерирует *Javadoc* проекта;

– *antrun* – запускает набор задач *ant* из любой указанной фазы.

Для понимания того, как это работает на практике, рассмотрим пример:

```
<profiles>
        <profile>
            <id>test</id>
            <activation>
                <file>
              <missing>target/generated sources/some/dir/com/iba/maven</missing>
                </file>
            </activation>
        </profile>
    </profiles>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
```

```
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-antrun-plugin</artifactId>
                <version>1.1</version>
                <executions>
                    <execution>
                        <id>id.clean</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>compile phase</echo>
                            </tasks>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Далее необходимо выполнить команду, как представлено на рисунке 1.17, и удостовериться в ее выполнении.

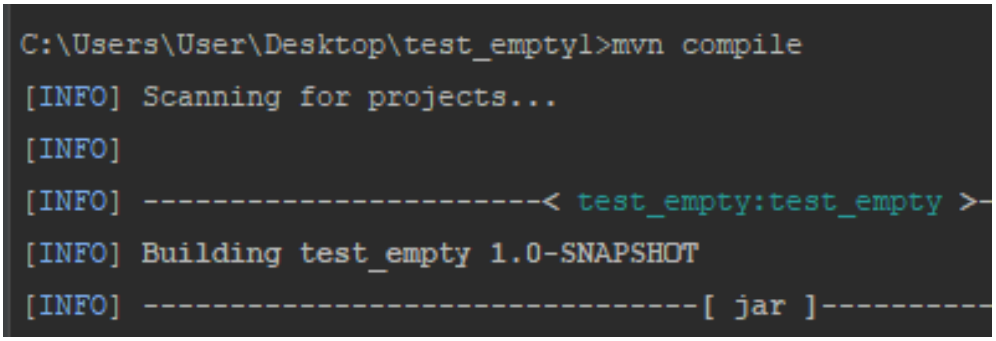


Рисунок 1.17 – Компиляция проекта с итегрированным плагином

1.8 Создание и сборка проекта с интеграцией зависимостей

Ранее мы уже изучали управление зависимостями с помощью репозиториев. Но что, если необходимые файлы не найдены ни в центральном, ни на удаленном репозитории? Для решения этой проблемы используются внешние зависимости.

Рассмотрим такой пример.

Добавим в наш проект в папку *src* директорию *lib*. Добавьте в эту директорию любой *jar*-файл (рисунок 1.18).

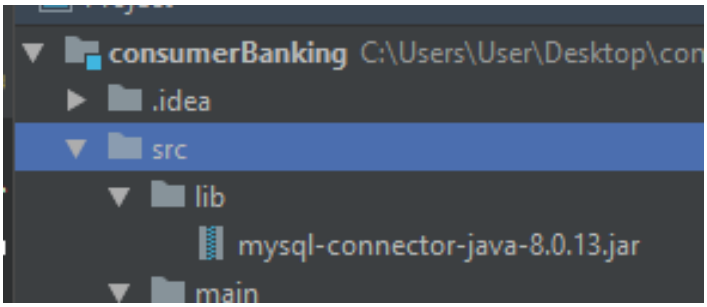


Рисунок 1.18 – Внешняя зависимость для подключения к проекту

После чего изменим наш *pom*-файл.

```
<profiles>
        <profile>
            <id>test</id>
            <activation>
                <file>
              <missing>target/generated sources/some/dir/com/iba/maven</missing>
                </file>
            </activation>
        </profile>
    </profiles>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <!-- External dependency -->
        <dependency>
            <groupId>mysql-connector-java-8.0.13.jar</groupId>
            <artifactId>mysql-connector-java-8.0.13.jar</artifactId>
            <scope>system</scope>
            <version>1.0</version>
            <systemPath>C:/Users/User/Desktop/consumerBanking/src/lib/mysql-
connector-java-8.0.13.jar</systemPath>
        </dependency>
    </dependencies>
```

После наших правок проект сможет обработать нашу библиотеку.

Тема 2. Системы контроля версий на примере *Git*

Git – система управления версиями с распределенной архитектурой. В отличие от некогда популярных систем вроде *CVS* и *Subversion (SVN)*, где полная история версий проекта доступна лишь в одном месте, в *Git* каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.

Разработка в *Git* ориентирована на обеспечение высокой производительности, безопасности и гибкости распределенной системы.

С помощью *Git* вы можете вернуть свой проект к более старой версии, сравнивать, анализировать или объединять свои изменения в репозиторий. Репозиторием называют хранилище вашего кода и историю его изменений. *Git* работает локально и все ваши репозитории хранятся в определенных папках на жестком диске.

Также ваши репозитории можно хранить и в интернете. Обычно для этого используют три сервиса:

– *GitHub*;
– *Bitbucket*;
– *GitLab*.

Каждая точка сохранения вашего проекта носит название коммит (*commit*). У каждого *commit* есть *hash* (уникальный *id*) и комментарий. Из таких *commit* собирается ветка. Ветка – это история изменений. У каждой ветки есть свое название. Репозиторий может содержать в себе несколько веток, которые создаются из других веток или вливаются в них.

2.1 Настройка рабочего пространства и регистрация в системе

Заведите бесплатную учетную запись на сайте *https://github.com/*. Регистрация на сайте заключается в выборе имени пользователя и пароля. Заполните информацию о себе на странице вашего профиля (рисунок 2.1).

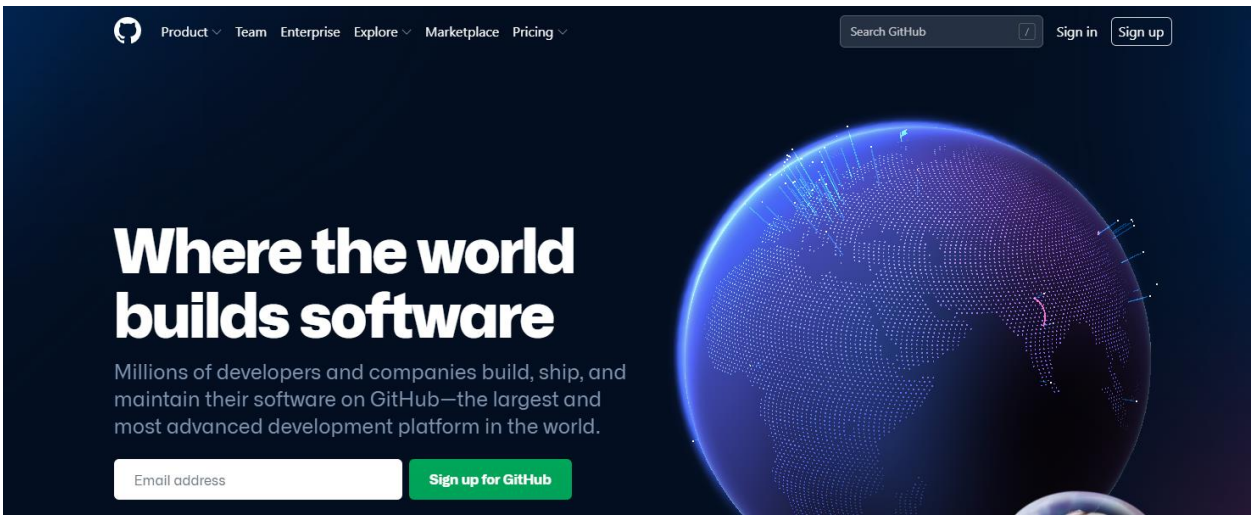


Рисунок 2.1 – Стартовая страница с регистрацией *GitHub*

Создайте репозиторий для вашего первого проекта (можно загрузить старый проект). Для этого нужно перейти по ссылке *https://github.com/new*, выбрать название репозитория (произвольным образом), описание и режим доступа *Public*. Выполните настройку согласно рисунку 2.2.

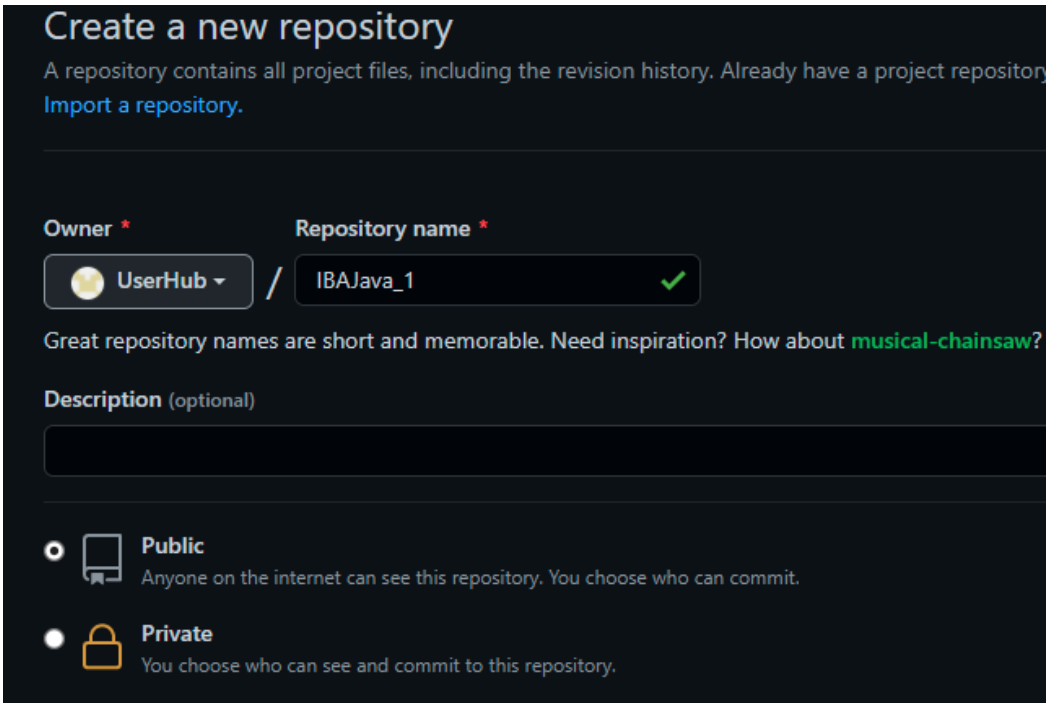


Рисунок 2.2 – Выбор настроек репозитория

В дальнейшем можно будет использовать разделы подсказок. Воспользуйтесь разделом «*…or create a new repository on the command line*» из

подсказки, чтобы инициализировать репозиторий и создать в нем файл *README.md* и т. д.

Мы создадим пустой репозиторий с файлом *README.md*. Для этого в шапке конфигурации нужно нажать соответствующую ссылку. Модифицируйте файл *README.md* так, чтобы он содержал ваше имя, номер группы. Добавляем его в проект (зеленая кнопка внизу) (рисунки 2.3 и 2.4).

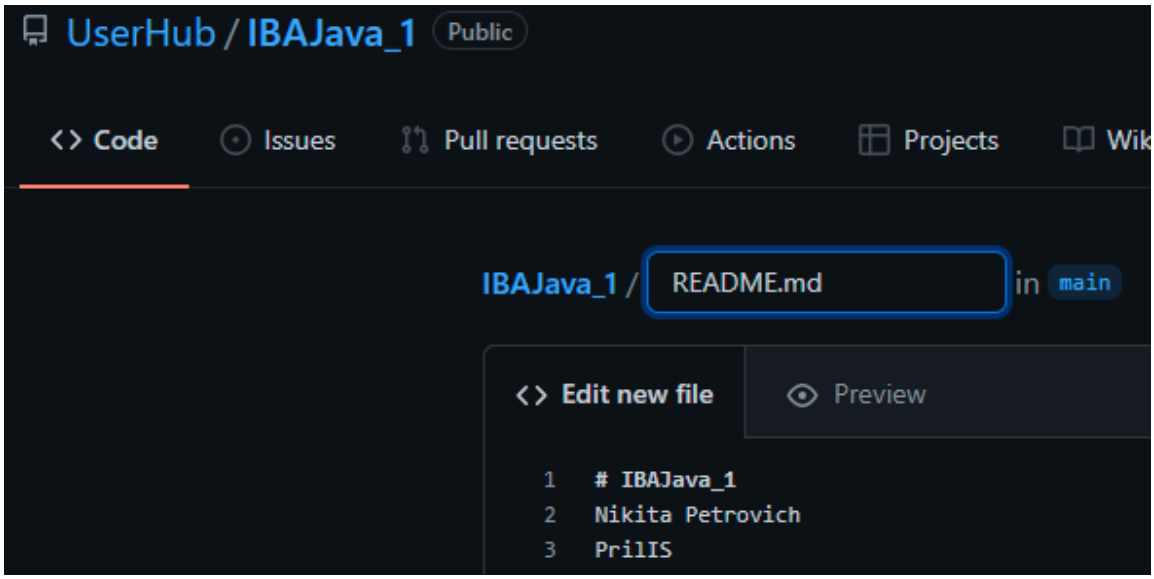


Рисунок 2.3 – Редактирование файлов репозитория

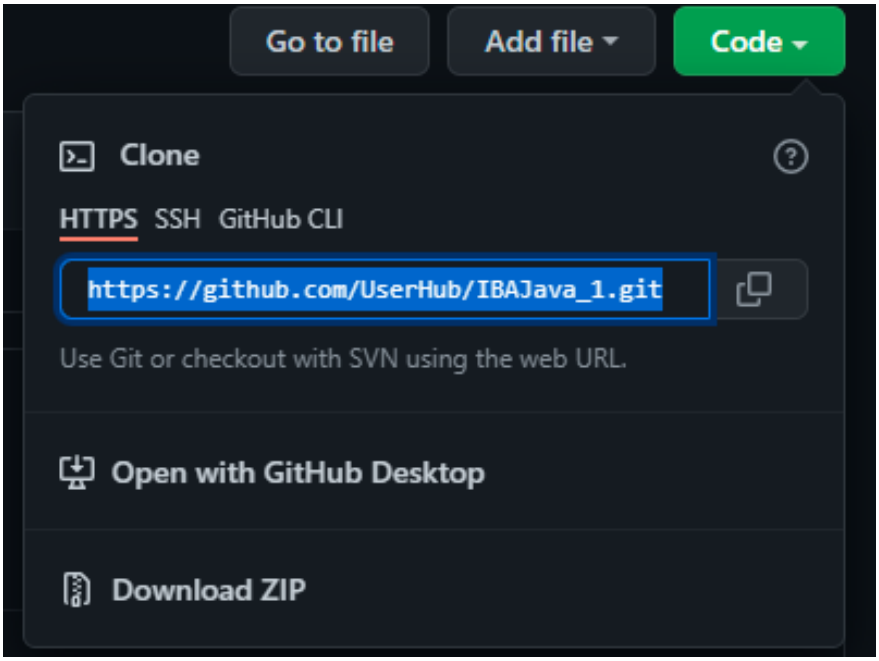


Рисунок 2.4 – Получение ссылки репозитория

2.2 Установка дистрибутива *Git*

Скачайте дистрибутив по ссылке *http://git-scm.com/download/* и установите консольного клиента (*gitbush*). Проверьте версию (рисунки 2.5 и 2.6).

Рисунок 2.5 – Выбор дистрибутива

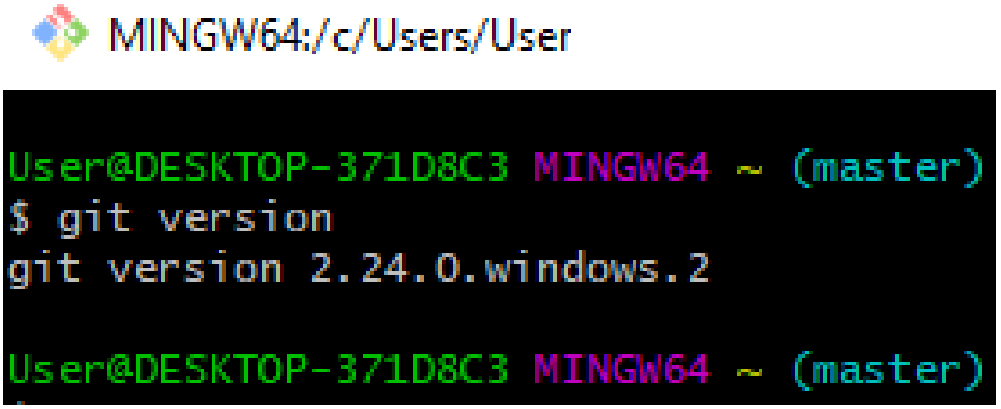


Рисунок 2.6 – Проверка версии системы *Git*

2.3 Настройка конфигурации системы контроля версии

Мы можем выбрать пустую папку или существующий проект. Нажать внутри правой кнопкой мыши и выбрать *Git Bash Here*. После чего откроется консоль для управления версиями в текущей директории.

Выполните конфигурацию (*git config*) с помощью команды *$ git config --global user.name "XXX"*, а также команды *$ git config --global user.email XXX@XXX.com* (настройте имя пользователя и т. п.) (рисунок 2.7).

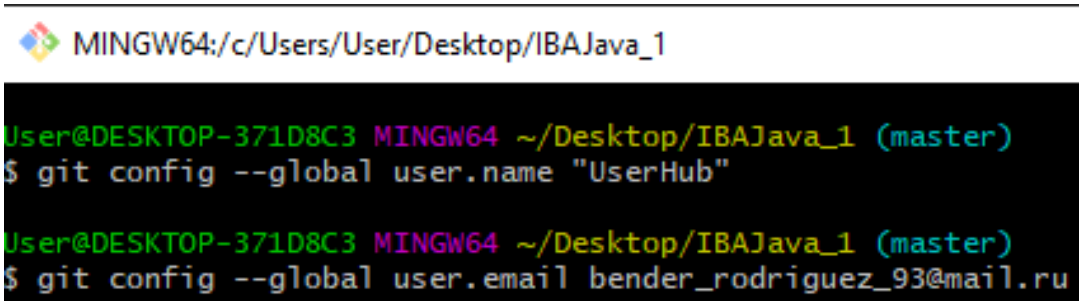


Рисунок 2.7 – Установка глобальных значений для системы контроля версии

Передача параметра *--global* позволяет сделать эти настройки всего один раз.

Если для конкретного проекта требуется указать другое имя или адрес электронной почты, войдите в папку с проектом и выполните эту команду без параметра *--global*.

Проверить выбранные настройки позволяет команда *git config --list*, выводящая список всех обнаруженных параметров (рисунок 2.8).

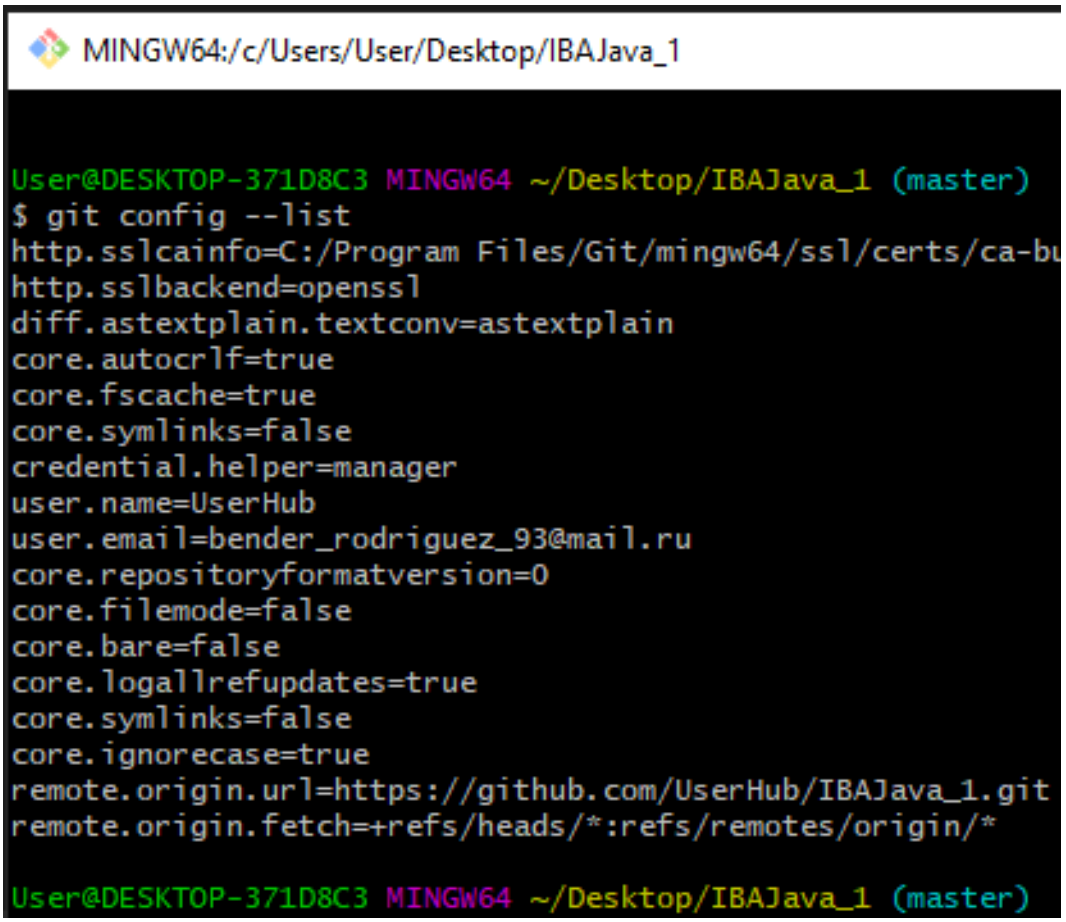


Рисунок 2.8 – Проверка настроек

2.4 Создание и инициализация репозиториев

Есть два подхода к созданию *Git*-проекта. Можно взять существующий проект или папку и импортировать в *Git*, а можно клонировать уже существующий репозиторий с другого сервера.

Чтобы начать слежение за существующим проектом, перейдите в папку этого проекта и введите команду согласно рисунку 2.9:

$ git init

Эта команда создает в текущей директории новую скрытую поддиректорию с именем *.git*, содержащую все необходимые файлы репозитория, – основу *Git*-репозитория. Если вы сделали ошибку, можно просто ее удалить (рисунок 2.9).

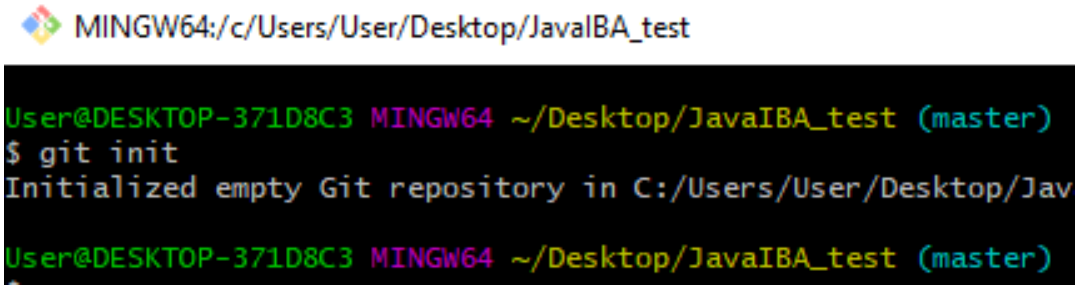


Рисунок 2.9 – Инициализация репозитория

Выполните:

$ git status

Если использовать пустой проект, то основным инструментом определения состояния файлов является команда *git status*. Она показывает, что изменения не зафиксированы (рисунок 2.10).

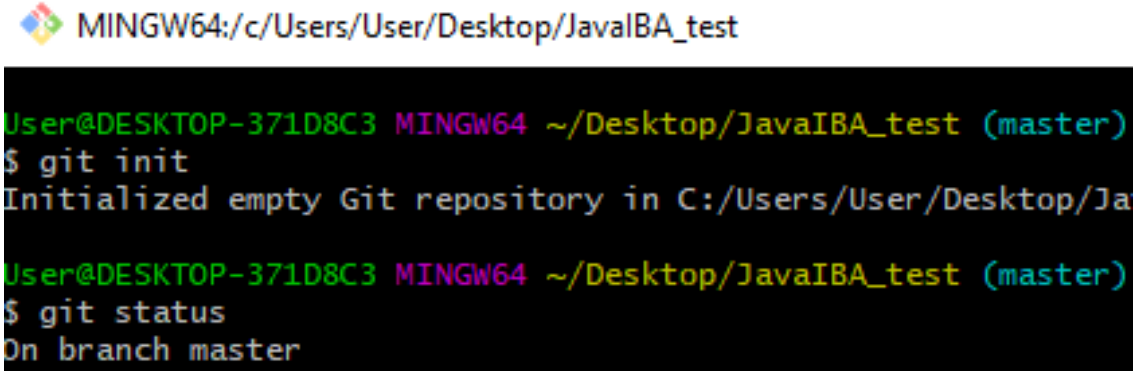


Рисунок 2.10 – Получение данных репозитория

При создании репозитория в существующем проекте обратите внимание на файлы, выделенные красным цветом (рисунок 2.11).

Рисунок 2.11 – Статус неотслеживаемых файлов

Статус *Untracked files*, по сути, означает, что *Git* видит файл, отсутствующий в предыдущем коммите (снимке состояния). *Git* не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите.

Выполните первую фиксацию изменений. Чтобы начать слежение за новым файлом, воспользуйтесь командой *git add*, добавляющей файлы, за которыми вы хотите следить.

Теперь команда *status* покажет, что этот файл является отслеживаемым и проиндексированным.

Затем следует команда *git commit*

*$ git add *.java*

$ git commit -m 'start commit'

Обратите внимание, что после добавления *add* все файлы из *src* пропали как неотслеживаемые (*modified* у вас не будет), (рисунки 2.12 и 2.13).

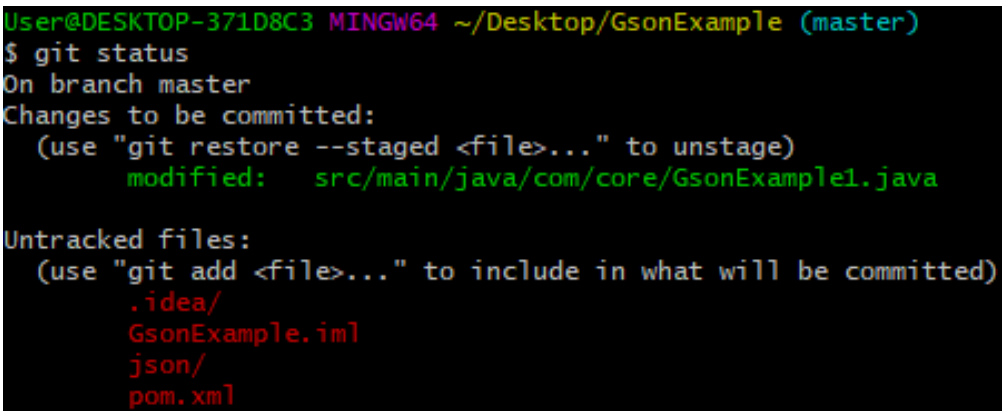


Рисунок 2.12 – Изменение отслеживаемого файла

```
User@DESKTOP-371D8C3 MINGW64 ~/Desktop/GsonExample (master)
$ git commit -m "start commit"
[master (root-commit) 7134d83] start commit
 7 files changed, 275 insertions(+)
 create mode 100644 src/main/java/com/core/CustomExclusionStrategy.java
 create mode 100644 src/main/java/com/core/ExcludeField.java
 create mode 100644 src/main/java/com/core/GsonExample1.java
 create mode 100644 src/main/java/com/core/GsonExample2.java
 create mode 100644 src/main/java/com/domain/Software.java
 create mode 100644 src/main/java/com/domain/Staff.java
 create mode 100644 src/test/java/com/tb/java/AppTest.java
```

Рисунок 2.13 – Фиксирование изменений

Сообщение фиксации можно задать и в команде *commit*, поставив перед ней флаг *-m* (как на рисунке 2.13). Вы создали первую версию изменений. Обратите внимание, что версия предоставила вам ряд данных о себе: зафиксированная вами ветка *master*, контрольная сумма *SHA-1* версии, количество подвергшихся изменениям файлов и статистика добавленных и удаленных строк

Внесем изменения в любой файл, находящийся под наблюдением. Отредактируйте отслеживаемый файл *GsonExample.java* (можете создать любой файл с любым содержимым для теста) и далее воспользуйтесь проверкой с помощью команды *git status* (рисунок 2.14).

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/main/java/com/core/GsonExample1.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/
        GsonExample.iml
        json/
        pom.xml
        target/

no changes added to commit (use "git add" and/or "git commit -a")
```

Рисунок 2.14 – Отслеживание изменений в тестовом коде

Оказывается, *Git* индексирует файл в том состоянии, в котором он пребывал на момент выполнения команды *git add*. Если сейчас зафиксировать изменения, в коммит войдет версия файла *GsonExample1.java*, появившаяся

после последнего запуска команды *git add*, а не версия, находившаяся в рабочей папке при запуске команды *git commit*. Редактирование файла после выполнения команды *git add* требует повторного запуска этой команды для индексирования самой последней версии файла.

В *Git* существует флаг, позволяющий получить сведения в более компактной форме. Запустив команду *git status -s* или *git status --short*, вы получите упрощенный вариант вывода (рисунок 2.15).

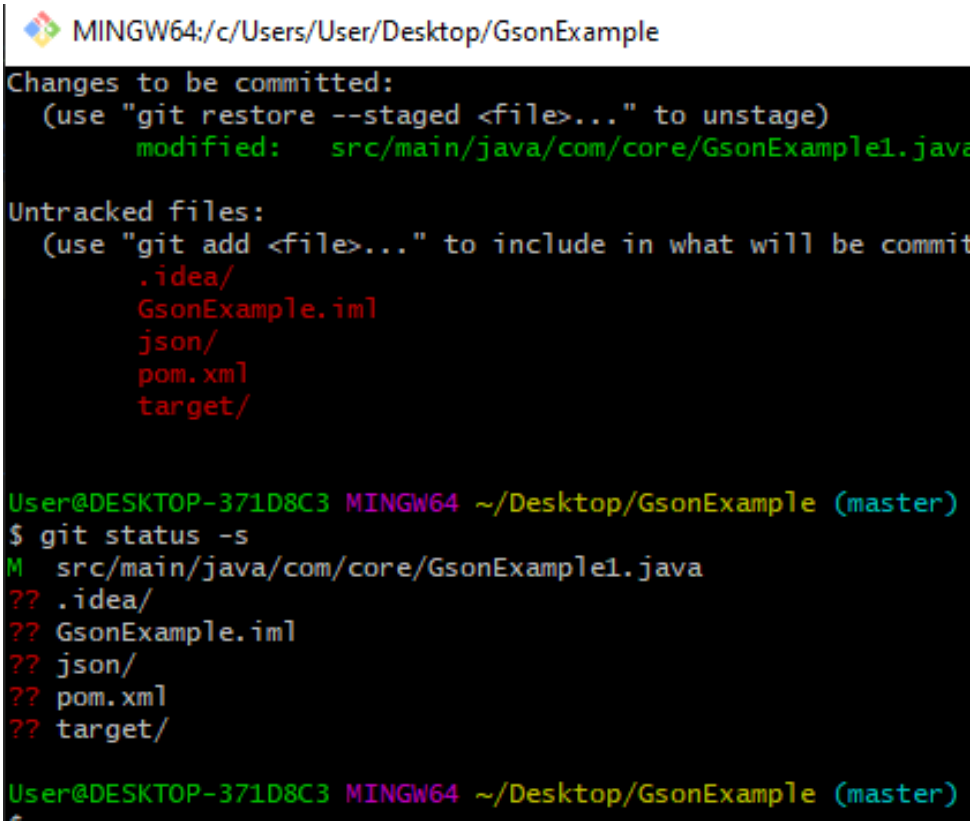


Рисунок 2.15 – Упрощенный вариант вывода информации

Рядом с именами новых неотслеживаемых файлов стоит знак «??», новые файлы, добавленные в область предварительной подготовки, помечены буквой A, а модифицированные файлы – буквой M.

2.5 Просмотр истории версий

После сохранения нескольких версий файлов вы, скорее всего, захотите взглянуть на то, что было сделано ранее. Базовым инструментом в данном случае является команда *git log* (рисунок 2.16).

Рисунок 2.16 – Проверка предудущих сохранений

Команда *git log* выводит в обратном хронологическом порядке список сохраненных в данный репозиторий версий, т. е. первыми показываются самые свежие коммиты. Как видите, рядом с каждым коммитом указывается его контрольная сумма *SHA-1*, имя и электронная почта автора, дата создания и сообщение о фиксации.

Если вы вдруг поняли, что не хотите сохранять внесенные в файл изменения, можно отменить их и вернуть файл в состояние, в котором он находился до последней фиксации.

Например, если мы внесли изменения в файл (см. рисунок 2.14), то получим следующий результат (рисунок 2.17).

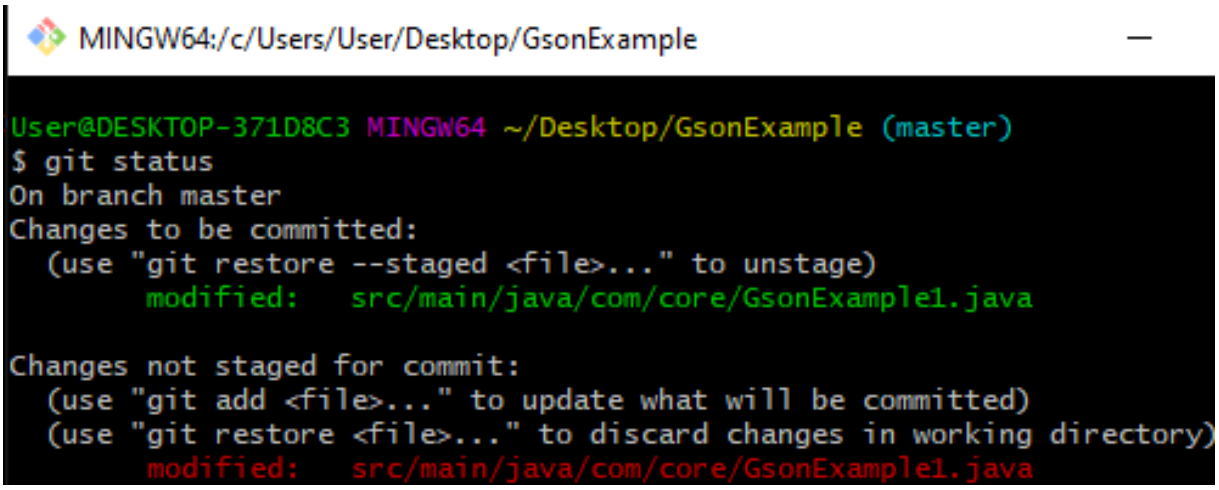


Рисунок 2.17 – Подтверждение изменений файлов

Если прочитать сообщения, то видно, что здесь указано, как сбросить принятые изменения (рисунок 2.18).

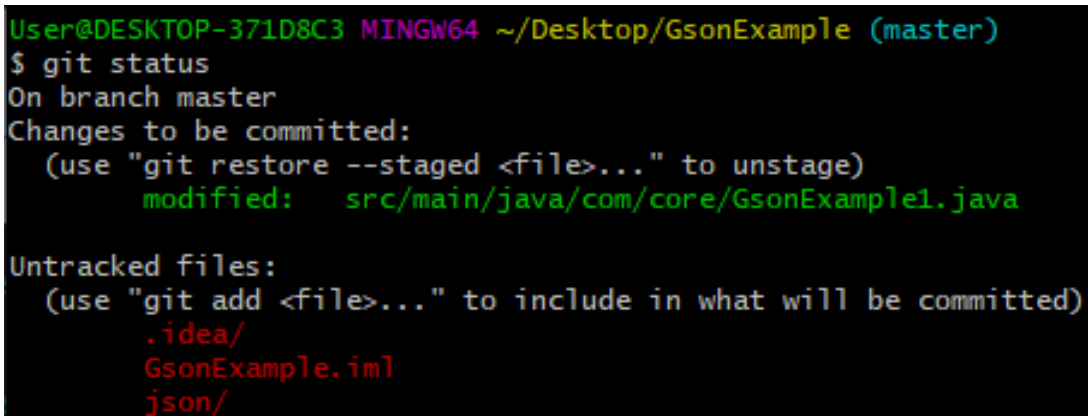


Рисунок 2.18 – Проверка возможности сбросить принятые изменения

Как видите, откат изменений выполнен. Важно понимать, что *git restore <file>* – опасная команда. Любые изменения соответствующего файла пропадают – вы просто копируете поверх него другой файл. Ни в коем случае не используйте эту команду, если вы не убеждены, что файл вам не нужен.

Все, что зафиксировано коммитом в *Git*, почти всегда можно восстановить. Если не сделать коммит, информация будет потеряна.

2.6 Клонирование существующего репозитория

Получение копии существующего репозитория, например проекта, в котором вы хотите принять участие, выполняется командой *git clone*.

Команда *git clone* по умолчанию забирает все версии всех файлов за всю историю проекта.

Клонирование репозитория осуществляется командой *git clone <url>* (рисунок 2.19).

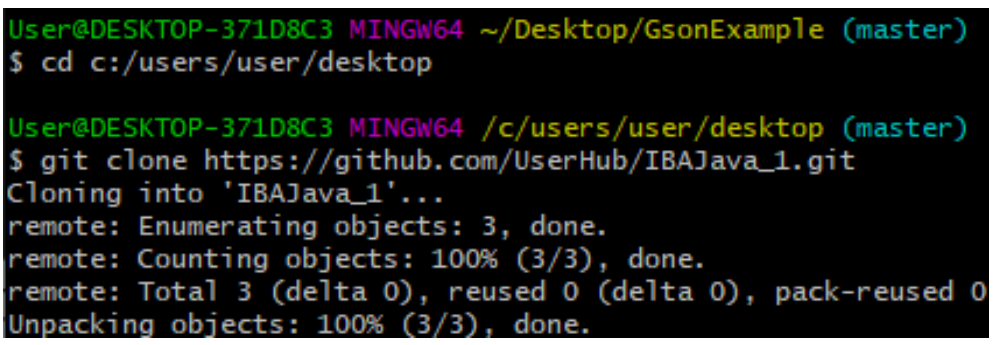


Рисунок 2.19 – Клонирование репозитория

Команда создает папку с именем *IBAJava_1*, инициализирует в ней папку *.git*, считывает из репозитория все данные и выгружает рабочую копию последней версии. В папке вы найдете все файлы проекта, подготовленные к работе.

2.7 Работа с удаленными репозиториями

Удаленные репозитории представляют собой версии вашего проекта, сохраненные в интернете или еще где-то в сети. У вас может быть несколько удаленных репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удаленными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории.

Просмотр уже настроенных удаленных серверов осуществляется командой *git remote*. Если репозиторий был клонирован, вы должны увидеть по крайней мере источник, т. е. имя, которое *Git* по умолчанию присваивает клонируемому серверу. Параметр *-v* позволяет увидеть *URL*-адреса, которые *Git* хранит для сокращенного имени, используемого при чтении из данного удаленного репозитория и при записи в него.

Извлечение данных из удаленных проектов выполняется командой

git fetch <имя удаленного репозитория>

Эта команда связывается с удаленным проектом и извлекает оттуда все пока отсутствующие у вас данные. После этого у вас должны появиться ссылки на все ветки удаленного проекта, которые можно подвергнуть слиянию или просмотреть.

При клонировании данная команда автоматически добавляет удаленный репозиторий под именем *origin*. Соответственно команда *git fetch origin* извлекает все, что появилось на этом сервере после его клонирования (или после момента последнего извлечения информации). Важно понимать, что команда *git fetch* помещает все данные в ваш локальный репозиторий. Она не выполняет автоматическое слияние с ветками, с которыми вы работаете в данный момент, и вообще никак не затрагивает эти ветки. Слияние вы выполните вручную, как только в этом возникнет необходимость (рисунок 2.20).

```
User@DESKTOP-371D8C3 MINGW64 /c/users/user/desktop/IBAJava_1 (main)
$ git fetch origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/UserHub/IBAJava_1
   eb25ffb..1f4c1a2  main       -> origin/main

User@DESKTOP-371D8C3 MINGW64 /c/users/user/desktop/IBAJava_1 (main)
$ git pull
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Рисунок 2.20 – Извлечение данных из удаленного репозитория

Обратите внимание, что произошел конфликт при выполнении команды. Если файл, который вы пытаетесь получить уже существует, а также данные вашего файла имеют недостающие фрагменты, которых нет в удаленном репозитории, то будет такая ошибка. Требуется его разрешить. Зайдем в файл в каталоге и исправим его при необходимости, т. е. выберем информацию, которая должна остаться в единственном экземпляре (рисунок 2.21).

README.md

```
1  # IBAJava_1
2  Nikita Petrovich
3  PrilIS
4  <<<<<<< HEAD
5
6  New test DATA
7  =======
8  try to add some info
9  >>>>>>> 1f4c1a2cf3a4595f2314f344d3c29d815f788563
```

Рисунок 2.21 – Пример конфликта сохраненных версий

Когда вы хотите поделиться своими наработками, вам необходимо отправить (*push*) их в главный репозиторий. Команда для этого действия простая: *git push <remote-name> <branch-name>*. Чтобы отправить вашу ветку *master* на сервер *origin* (повторимся, что клонирование, как правило, настраивает оба этих

имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

$ git push origin master

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и, если никто другой с тех пор не выполнял команду *push*. Если вы и кто-то еще одновременно клонируете, затем он выполняет команду *push*, а затем команду *push* выполняете вы, то ваш *push* точно будет отклонен. Вам придется сначала получить (*pull*) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить *push*.

Тема 3. Управление персистенцией на основе *JPA*

В прежнее время люди хранили данные просто в файлах. Но как только возникла необходимость одновременного доступа к чтению и редактированию, с появлением нагрузки (т. е. одновременно поступает несколько обращений), хранение данных просто в файлах становится проблемой.

С давних пор *Java* умеет работать с базами данных (БД) при помощи *JDBC API (The Java Database Connectivity)*.

Со временем разработчики каждый раз сталкивались с необходимостью писать однотипный и ненужный «обслуживающий» код (так называемый *Boilerplate code*) для тривиальных операций по сохранению *Java*-объектов в БД и наоборот, созданию *Java*-объектов по данным из БД. И тогда для решения этих проблем появилось такое понятие, как *ORM*.

ORM – Object-Relational Mapping или в переводе на русский «объектно-реляционное отображение». Это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования.

ORM – это, по сути, концепция о том, что *Java*-объект можно представить как данные в БД. Она воплотилась в виде спецификации *JPA – Java Persistence API*.

Спецификация – это уже описание *Java API*, которое выражает эту концепцию. Спецификация рассказывает, какими средствами мы должны быть обеспечены (т. е. через какие интерфейсы мы сможем работать), чтобы работать по концепции *ORM*, и как использовать эти средства.

Реализацию средств спецификация не описывает. Это дает возможность использовать для одной спецификации разные реализации. Можно упростить и сказать, что спецификация – это описание *API*.

Текст спецификации *JPA* можно найти в одноименном разделе на сайте *Oracle*: *«JSR 338: JavaTM Persistence API»*.

Следовательно, чтобы использовать *JPA* нам требуется некоторая реализация, при помощи которой мы будем пользоваться технологией.

3.1 Проектирование структуры проекта

Задача – реализовать часть бизнес-логики приложения управления магазином автомобилей. Возможность добавлять марки машин, осуществлять поиск в системе и *CRUD*-операции с компанией. Обеспечить наличие ролей пользователей в системе.

Нам потребуется следующая структура (рисунок 3.1).

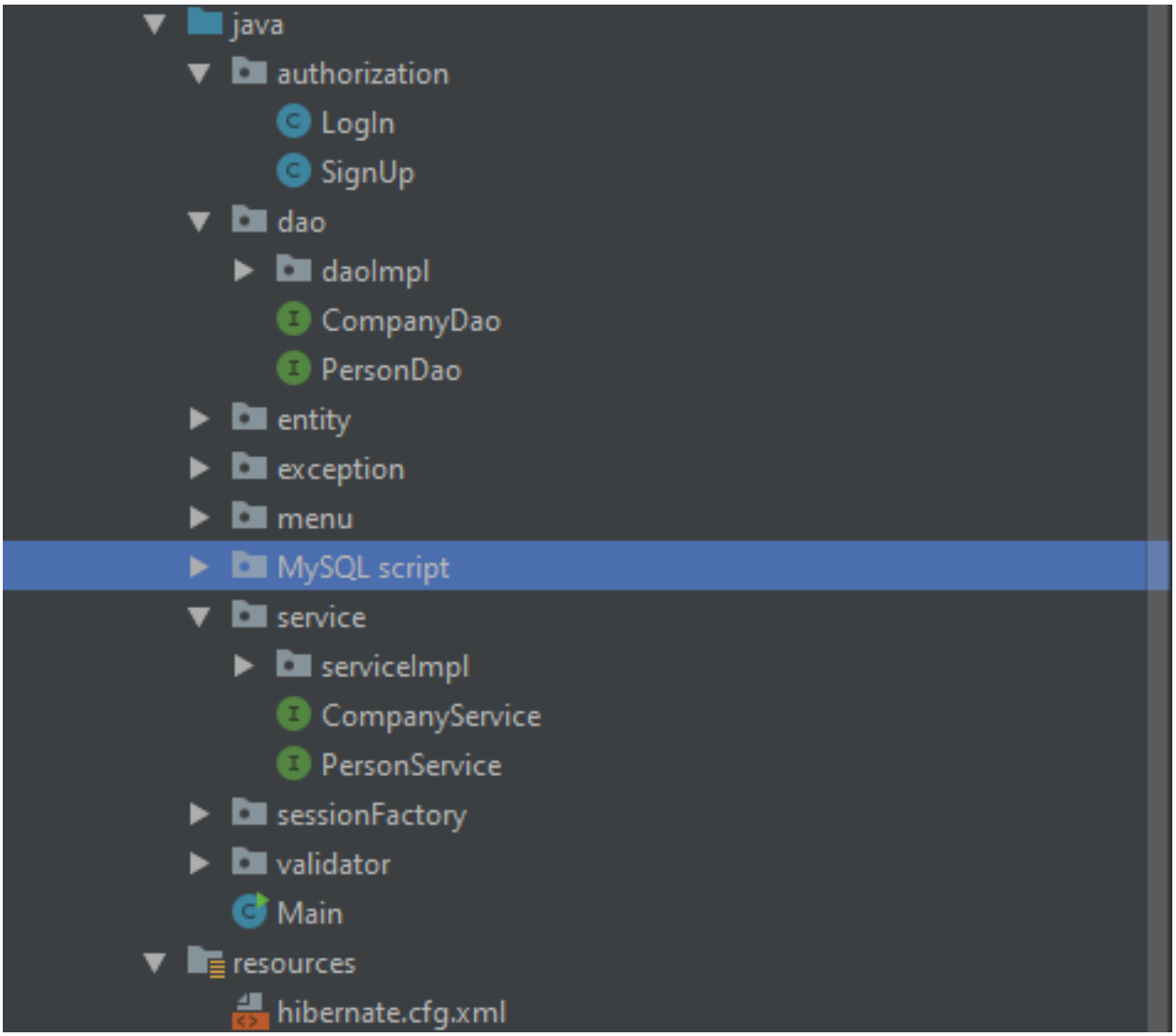


Рисунок 3.1 – Структура проекта

Начнем с конфигурации проекта.

Создайте пустой проект с помощью *Maven*. Далее откроем *pom* и воспользуемся зависимостями.

Добавить в *Properties*:

```
<hibernate-version>5.0.1.Final</hibernate-version>
<dependencies>
    <!--driver for connection to MYSql database -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
    <!-- Hibernate -->
    <!-- to start need only this -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.28.Final</version>
    </dependency>
    <!-- for JPA, use hibernate-entitymanager instead of hibernate-core ->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${hibernate-version}</version>
    </dependency>
</dependencies>
```

После чего нам необходимо создать папку ресурсов и добавить конфигурацию подключения нашей БД к проекту.

Файл *hibernate.cfg.xml*:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/carshop</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
        <property name="hibernate.connection.pool_size">1</property>
        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

       <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

```
        <property
name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.connection.CharSet">utf8</property>
        <property name="hibernate.connection.characterEncoding">utf8</property>
        <property name="hibernate.connection.useUnicode">true</property>
    </session-factory>
</hibernate-configuration>
```

Следующий файл – это класс *Main*.

```
import menu.Menu;
public class Main {
    public static void main(String[] args) {
        Menu menu = new Menu();
        menu.IntroducingMenu();
    }
}
```

Здесь мы сразу перенаправляем вызов на класс исполнителя. Методом хорошей практики считается не загружать класс *Main* кодом бизнес-логики.

Поработаем с частью системы в пакете *Menu* (рисунок 3.2).

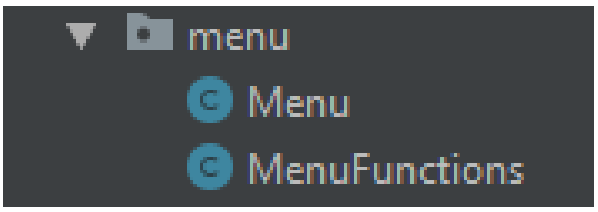


Рисунок 3.2 – Классы пользовательских функций

Наш класс *Main* перенаправил нас на метод класса *Menu*. Этот класс будет определять наборы меню для пользователей и вызов определенных методов – это первый слой нашей архитектуры, предназначенный для описания пользовательской информации и взаимодействия.

Нас в первую очередь интересует самый первый метод, который вызывается из *Main*, остальное чуть позже.

Рассмотрим, что он содержит:

```
package menu;
import authorization.LogIn;
import authorization.SignUp;
import entity.User;
import java.util.Scanner;
```

```
public class Menu {

    public Menu() { }
    Scanner in = new Scanner(System.in);
    MenuFunctions menuFunctions = new MenuFunctions();
    public void IntroducingMenu() {
        LogIn logIn = new LogIn();
        SignUp signUp = new SignUp();
        String choice = "0";
        while (Integer.parseInt(choice) != 3) {
            String s = "Меню\n" +
                    "1. Войти\n" +
                    "2. Зарегистрироваться\n" +
                    "3. Выход\n" +
                    "Выбор: ";
            System.out.print(s);
            choice = in.nextLine();
            switch (choice) {
                case "1":
                    logIn.authorization();
                    break;
                case "2":
                    signUp.registration();
                    break;
                case "3":
                    System.out.println("До свидания!");
                    break;
                default:
                    System.out.println("Проверьте корректность ввода!");
                    choice = "0";
                    break;
            }
        }
    }
    public void AdminMenu() {
        String choice = "0";
        while (Integer.parseInt(choice) != 4) {
            String s = "Меню админа\n" +
                    "1. Работа с пользователями\n" +
                    "2. Работа с компаниями\n" +
                    "3. Работа с машинами\n" +
                    "4. Выход\n" +
                    "Выбор: ";
            System.out.print(s);
            choice = in.nextLine();

  switch (choice) {
                case "1":
                    AdminMenuWithUsers();
                    break;
                case "2":
                    AdminMenuWithCompanies();
                    break;
```

```
            case "3":
                AdminMenuWithCars();
                break;
            case "4":
                break;
            default:
                System.out.println("Проверьте корректность ввода!");
                choice = "0";
                break;
        }
    }
}

public void UserMenu(User currentUser) {
    String choice = "0";
    while (Integer.parseInt(choice) != 6) {
        String s = "Меню пользователя\n" +
                "1. Просмотреть все компании\n" +
                "2. Просмотреть все модели машин\n" +
                "3. Найти компанию по названию\n" +
                "4. Найти модель машины по названию\n" +
                "5. Отредактировать свой профиль\n" +
                "6. Выход\n" +
                "Выбор: ";
        System.out.print(s);
        choice = in.nextLine();
        switch (choice) {
            case "1":
                menuFunctions.showCompanies();
                break;
            case "2":
                menuFunctions.showCars();
                break;
            case "3":
                menuFunctions.showOneCompany();
                break;
            case "4":
                menuFunctions.findCarByName();
                break;
            case "5":
                menuFunctions.updateLoginAndPassword(currentUser);
                break;
            case "6":
                break;
            default:
                System.out.println("Проверьте корректность ввода!");
                choice = "0";
                break;
        }
    }
}
```

```
private void AdminMenuWithUsers() {
    String choice = "0";
    while (Integer.parseInt(choice) != 5) {
        String s = "Работа с пользователями\n" +
                "1. Добавить пользователя\n" +
                "2. Изменить личные данные пользователя\n" +
                "3. Удалить пользователя\n" +
                "4. Просмотреть всех пользователей\n" +
                "5. Выход\n" +
                "Выбор: ";
        System.out.print(s);
        choice = in.nextLine();
        switch (choice) {
            case "1":
                menuFunctions.addPerson();
                break;
            case "2":
                menuFunctions.updatePerson();
                break;
            case "3":
                menuFunctions.deletePerson();
                break;
            case "4":
                menuFunctions.showPeople();
                break;
            case "5":
                break;
            default:
                System.out.println("Проверьте корректность ввода!");
                choice = "0";
                break;
        }}}
public void AdminMenuWithCompanies() {
    String choice = "0";
    while (Integer.parseInt(choice) != 5) {
        String s = "Работа с компаниями\n" +
                "1. Добавить компанию\n" +
                "2. Изменить компанию\n" +
                "3. Удалить компанию\n" +
                "4. Вывести все компании\n" +
                "5. Выход\n" +
                "Выбор: ";
        System.out.print(s);
        choice = in.nextLine();
        switch (choice) {
            case "1":
                System.out.println(menuFunctions.addCompany());
                break;
            case "2":
                menuFunctions.updateCompany();
                break;
            case "3":
                menuFunctions.deleteCompany();
```

```
                    break;
                case "4":
                    menuFunctions.showCompanies();
                    break;
                case "5":
                    break;
                default:
                    System.out.println("Проверьте корректность ввода!");
                    choice = "0";
                    break;
            }}
    }
    public void AdminMenuWithCars() {
        String choice = "0";
        while (Integer.parseInt(choice) != 7) {
            String s = "Работа с машинами\n" +
                    "1. Добавить машину\n" +
                    "2. Изменить машину\n" +
                    "3. Удалить машину\n" +
                    "4. Вывести все машины\n" +
                    "5. Вывести все машины одной компании\n" +
                    "6. Найти машину по названию\n" +
                    "7. Выход\n" +
                    "Выбор: ";
            System.out.print(s);
            choice = in.nextLine();
            switch (choice) {
                case "1":
                    menuFunctions.addCar();
                    break;
                case "2":
                    menuFunctions.updateCar();
                    break;
                case "3":
                    menuFunctions.deleteCar();
                    break;
                case "4":
                    menuFunctions.showCars();
                    break;
                case "5":
                    menuFunctions.showCarsFromOneCompany();
                    break;
                case "6":
                    menuFunctions.findCarByName();
                case "7":
                    break;
                default:
                    System.out.println("Проверьте корректность ввода!");
                    choice = "0";
                    break;
            }
    }}
}
```

Как мы видим из данного класса, есть три возможности: получить доступ, зарегистрироваться и выйти. При выборе пункта меню нам потребуется вызвать методы класса входа в систему и регистрации.

Рассмотрим часть входа и регистрации (рисунок 3.3).

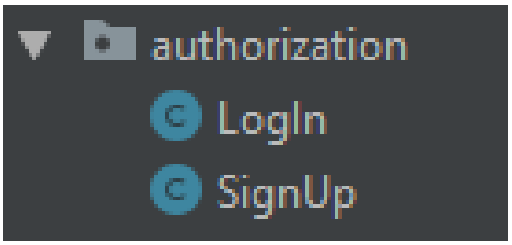


Рисунок 3.3 – Классы входа и регистрации пользователей

Класс *LogIn* имеет следующую структуру:

```
package authorization;
import entity.Person;
import entity.User;
import menu.Menu;
import service.PersonService;
import service.serviceImpl.PersonServiceImpl;
import java.util.List;
import java.util.Scanner;

public class LogIn {

    Scanner in = new Scanner(System.in);

    public void authorization() {
        PersonService personService = new PersonServiceImpl();
        List<Person> people = personService.showPeople();
        System.out.print("Введите логин: ");
        String login = in.nextLine();
        System.out.print("Введите пароль: ");
        String password = in.nextLine();
        User currentUser = null;
        for(Person p : people) {
            if(p.getUser().getLogin().equals(login) &&
p.getUser().getPassword().equals(password)) {
                currentUser = p.getUser();
                p.setPersonId(people.size());
            }
            if (p.getUser().getLogin().equals(login) &&
!p.getUser().getPassword().equals(password)) {
                System.out.println("Проверьте корректность пароля!");
            }
        }
        if (currentUser != null) {
```

```
            System.out.println("Авторизация пройдена успешно! Добро пожаловать "
+
                    currentUser.getPerson().getSurname() + " " +
currentUser.getPerson().getName());
            Menu menu = new Menu();
            String role = currentUser.getRole();
            switch (role) {
                case "Admin":
                    menu.AdminMenu();
                    break;
                case "User":
                    menu.UserMenu(currentUser);
                    break;
            }
        }
        else {
            System.out.println("Такого пользователя не найдено");
        }
    }
}
```

И класс *SignUp*:

```
package authorization;
import menu.MenuFunctions;

public class SignUp {

    public void registration() {
        MenuFunctions menuFunctions = new MenuFunctions();
        menuFunctions.addPerson();
    }
}
```

Начнем с *SignUp*.

Здесь мы видим создание объекта другого класса и перенаправление к нему. Вопрос: почему нельзя это было сделать в предыдущем классе? Зачем делать промежуточный класс? Ответ: во всех языках придерживаются правил написания. Существуют правила, как официальные, так и неофициальные. Например, самый распространенный стек правил, который вы могли слышать, – *SOLID*, который устанавливает правильность или грамотность создания кода (если говорить утрированно). Также можно привести *CodConventions* и т. д.

Чтобы закрыть раздел с *Menu,* добавим код для классов *MenuFunctions* и рассмотрим его подробнее. Нас будет пока интересовать только первый метод, но остальные методы тоже кратко рассмотрим, чтобы представлять назначение класса. Более подробно их разберем при соответствующих вызовах.

```
package menu;
import entity.Car;
import entity.Company;
import entity.Person;
import entity.User;
import service.CompanyService;
import service.PersonService;
import service.serviceImpl.CompanyServiceImpl;
import service.serviceImpl.PersonServiceImpl;
//import validator.Validator;

import java.util.List;
import java.util.Scanner;

public class MenuFunctions {

    Scanner in = new Scanner(System.in);
    PersonService personService = new PersonServiceImpl();
    CompanyService companyService = new CompanyServiceImpl();
    public MenuFunctions() { }

    public void addPerson() {
        System.out.println("---Добавление пользователя---");
        Person person = getPersonInfo();
        if (person != null) {
            if (personService.addPerson(person)) {
                System.out.println("---Добавление выполнено!---");
            }
        }
    }
    private Person getPersonInfo() {
        Person person = null;
        System.out.print("Введите имя пользователя: ");
        String name = in.nextLine();
        System.out.print("Введите фамилию пользователя: ");
        String surname = in.nextLine();
        System.out.print("Введите возраст пользователя: ");
        String age = in.nextLine();
        System.out.print("Введите телефон пользователя: ");
        String phone = in.nextLine();
        System.out.print("Введите почту пользователя: ");
        String mail = in.nextLine();
        //if (Validator.correctPerson(name, surname, age, phone, mail)) {
            User user = getUserInfo();
            if (user != null) {
                person = new Person(name, surname, Integer.parseInt(age), phone,
mail);
                user.setPerson(person);
                person.setUser(user);
            }
            else {
                System.out.println("Пароль или логин некорректны!");
            }
      /*  }
```

```
    else {
        System.out.println("Личные данные некорректны!");
    }*/
    return person;
}

private User getUserInfo() {
    User user = null;
    System.out.print("Введите логин пользователя: ");
    String login = in.nextLine();
    System.out.print("Введите пароль пользователя: ");
    String password = in.nextLine();
  //  if(Validator.correctUser(login, password)) {
        if(checkUniqueLogin(login)) {
            user = new User(login, password, "User");
        }
        else {
            System.out.println("Такой логин уже занят!");
        }
   //}
    return user;
}

private boolean checkUniqueLogin(String login) {
    boolean isUnique = true;
    for (Person p : getPeople()) {
        if (p.getUser().getLogin().equals(login)) {
            isUnique = false;
        }
    }
    return isUnique;
}

public void updatePerson() {
    System.out.println("---Изменение пользователя---");
    showPeople();
    System.out.print("Выберите ID пользователя для изменения: ");
    String id = in.nextLine();
    if (getPersonId(id)) {
        Person person = findPersonById(Integer.parseInt(id));
        if (person != null) {
            changeDataFromPerson(person);
            changeDataFromUser(person);
            if (personService.updatePerson(person)) {
                System.out.println("---Изменение выполнено!---");
            }
        }
    }
}

private Person changeDataFromPerson(Person person) {
    System.out.print("Введите имя пользователя: ");
    String name = in.nextLine();
    System.out.print("Введите фамилию пользователя: ");
```

```
    String surname = in.nextLine();
    System.out.print("Введите возраст пользователя: ");
    String age = in.nextLine();
    System.out.print("Введите телефон пользователя: ");
    String phone = in.nextLine();
    System.out.print("Введите почту пользователя: ");
    String mail = in.nextLine();
  //  if (Validator.correctPerson(name, surname, age, phone, mail)) {
        person.setName(name);
        person.setSurname(surname);
        person.setAge(Integer.parseInt(age));
        person.setPhone(phone);
        person.setMail(mail);
   /* }
    else {
        System.out.println("Личные данные некорректны!");
    }*/
    return person;
}
private Person changeDataFromUser(Person person) {

    System.out.print("Введите логин пользователя: ");
    String login = in.nextLine();
    System.out.print("Введите пароль пользователя: ");
    String password = in.nextLine();
 //   if(Validator.correctUser(login, password)) {
        person.getUser().setLogin(login);
        person.getUser().setPassword(password);
   // }
    return person;
}
public void updateLoginAndPassword(User user) {
    System.out.println("---Изменение логина и пароля---");
    changeDataFromUser(user.getPerson());
    if (personService.updatePerson(user.getPerson())) {
        System.out.println("---Изменение выполнено!---");
    }

}
public void deletePerson() {
    System.out.println("---Удаление пользователя---");
    showPeople();
    System.out.print("Выберите ID пользователя для изменения: ");
    String id = in.nextLine();
    if (getPersonId(id)) {
        if (personService.deletePerson(Integer.parseInt(id))) {
            System.out.println("---Удаление выполнено!---");
        }
    }
}

private boolean getPersonId(String id) {
    boolean isAppropriateNumber = false;
 //   if (Validator.correctId(id)) {
```

```
            if (!(Integer.parseInt(id) < 0) && !(Integer.parseInt(id) >
getPeople().size())) {
                isAppropriateNumber = true;
            }
            else {
                System.out.println("Такого ID нет!");
            }
        /* }
         else {
            System.out.println("ID некорректно!");
         }*/
         return isAppropriateNumber;
    }

    public void showPeople() {
        List<Person> people = getPeople();
        if (people.size() != 0) {
            System.out.format("%10s%20s%20s%10s%20s%30s%20s", "ID |", "Имя |",
"Фамилия |", "Возраст |", "Телефон |", "Почта |", "Логин |");
            for (Person p: people) {
                System.out.println(" ");
                System.out.format("%10s%20s%20s%10s%20s%30s%20s",
p.getPersonId() + " |", p.getName() + " |",
                                  p.getSurname() + " |", p.getAge() + " |",
                                  p.getPhone() + " |", p.getMail() + " |",
p.getUser().getLogin() + " |");

            }
            System.out.println(" ");
        }
        else {
            System.out.println("Нет пользователей!");
        }
    }

    private List<Person> getPeople() {
        List<Person> people = personService.showPeople();
        return people;
    }

    private Person findPersonById(int id) {
        Person person = personService.findPersonById(id);
        return person;
    }

    //---COMPANY---//

    public String addCompany() {
        System.out.println("---Добавление компании---");
        String result = null;
        System.out.print("Введите название компании: ");
        String name = in.nextLine();
        System.out.print("Введите страну происхождения компании: ");
        String country = in.nextLine();
```

```
      //  if (Validator.correctCompany(name, country)) {
            Company company = new Company(name, country);
            if (companyService.addCompany(company)) {
                result = "---Добавление выполнено!---";
            }
      /*  }
        else {
            result = "Данные некорректны!";
        }*/
        return result;
    }
    public String updateCompany() {
        String result = null;
        System.out.println("---Изменение компании---");
        showCompanies();
        System.out.print("Выберите ID компании для изменения: ");
        String id = in.nextLine();
        if (getCompanyId(id)) {
            System.out.print("Введите название компании: ");
            String name = in.nextLine();
            System.out.print("Введите страну происхождения компании: ");
            String country = in.nextLine();
       //     if (Validator.correctCompany(name, country)) {
              Company company =
companyService.findCompanyById(Integer.parseInt(id));
              company.setCompanyName(name);
              company.setCompanyCountry(country);
              if (companyService.updateCompany(company)) {
                  System.out.println("---Изменение выполнено!---");
              }
          /*  }
            else {
                System.out.println("Данные некорректны!");
            }*/
        }

        return result;
    }

    public void deleteCompany() {
        System.out.println("---Удаление компании---");
        showCompanies();
        System.out.print("Введите ID компании для удаления: ");
        String id = in.nextLine();
        if (getCompanyId(id)) {
            if(companyService.deleteCompany(Integer.parseInt(id))) {
                System.out.println("---Удаление выполнено!---");
            }
        }
    }

    private boolean getCompanyId(String id) {
        boolean isAppropriateNumber = false;
     //   if (Validator.correctId(id)) {
```

```
            if (!(Integer.parseInt(id) < 0) && !(Integer.parseInt(id) >
getCompanies().size())) {
                isAppropriateNumber = true;
            }
            else {
                System.out.println("Такого ID нет!");
            }
       /* }
        else {
            System.out.println("ID некорректно!");
        }*/
        return isAppropriateNumber;
    }
    public void showCompanies() {
        List<Company> companies = getCompanies();
        if (companies.size() != 0) {
            theHeaderForCompany();
            for (Company c: companies) {
                theTableForCompany(c);
            }
            System.out.println(" ");
        }
        else {
            System.out.println("Нет компаний!");
        }
    }
    private void theTableForCompany(Company c) {
        System.out.println(" ");
        System.out.format("%10s%20s%30s", c.getCompanyId() + " |",
c.getCompanyName() + " |", c.getCompanyCountry() + " |");
        System.out.println(" ");
    }

    private void theHeaderForCompany() {
        System.out.format("%10s%20s%30s"," ID |", "Название |", "Страна
происхождения |");
    }

    private List<Company> getCompanies() {
        List<Company> companies = companyService.showCompanies();
        return companies;
    }

    public void showOneCompany() {
        Company company = findCompanyByName();
        if (company != null) {
            theHeaderForCompany();
            theTableForCompany(company);
        }
    }

    private Company findCompanyByName() {
        System.out.print("Введите название компании: ");
        String name = in.nextLine();
```

```
        boolean isFound = false;
        for (Company c : getCompanies()) {
            if (c.getCompanyName().equals(name)) {
                isFound = true;
            }
        }
        Company company = null;
        if (isFound) {
            company = companyService.findCompanyByName(name);
        }
        else {
            System.out.println("Такой компании не найдено!");
        }
        return company;
    }

    //---CAR---//

    public void addCar() {
        System.out.println("---Добавление машины---");
        showCompanies();
        System.out.print("Выберите ID компании для авто: ");
        String id = in.nextLine();
        Car car = getCarInfo();
        if (car != null) {
            Company company =
companyService.findCompanyById(Integer.parseInt(id));
            car.setCompany(company);
            company.addCar(car);
            if (companyService.updateCompany(company)) {
                System.out.println("---Добавление выполнено!---");
            }
        }
    }
    private Car getCarInfo() {
        System.out.print("Введите название: ");
        String name = in.nextLine();
        System.out.print("Введите год создания: ");
        String year = in.nextLine();
        System.out.print("Введите пробег: ");
        String distance = in.nextLine();
        System.out.print("Введите вид топлива: ");
        String fuel = in.nextLine();
        System.out.print("Введите расход: ");
        String fuelConsumption = in.nextLine();
        System.out.print("Введите цену: ");
        String price = in.nextLine();
       // if (Validator.correctCar(name, year, distance, fuel, fuelConsumption,
price)) {
       //     if (Validator.correctFuel(fuel)) {
                Car car = new Car();
                car.setName(name);
                car.setYear(Integer.parseInt(year));
                car.setDistance(Integer.parseInt(distance));
```

```
                car.setFuel(fuel);
                car.setFuelConsumption(fuelConsumption);
                car.setPrice(Integer.parseInt(price));
                return car;
          /*  }
            else {
                System.out.println("Введите топливо: Бензин или Дизель!");
            }*/
       /* }
        else {
            System.out.println("Данные некорректны!");
        }*/
    }
    public void deleteCar() {
        System.out.println("---Удаление машины---");
        showCars();
        Company company = findCompanyByName();
        if (company != null) {
            System.out.print("Введите название машины: ");
            String name = in.nextLine();
            Car car = findCarInList(company, name);
            if (car != null) {
                company.getCars().remove(car);
                if (companyService.updateCompany(company)) {
                    System.out.println("---Удаление выполнено!---");
                }
            }
        }
    }
    public void updateCar() {
        System.out.println("---Изменение машины---");
        showCars();
        Company company = findCompanyByName();
        if (company != null) {
            System.out.print("Введите название машины: ");
            String nameForChange = in.nextLine();
            Car car = findCarInList(company, nameForChange);
            if (car != null) {
                System.out.print("Введите название: ");
                String name = in.nextLine();
                System.out.print("Введите год создания: ");
                String year = in.nextLine();
                System.out.print("Введите пробег: ");
                String distance = in.nextLine();
                System.out.print("Введите вид топлива: ");
                String fuel = in.nextLine();
                System.out.print("Введите расход: ");
                String fuelConsumption = in.nextLine();
                System.out.print("Введите цену: ");
                String price = in.nextLine();
              /*  if (Validator.correctCar(name, year, distance, fuel,
fuelConsumption, price)) {
                    if (Validator.correctFuel(fuel)) {*/
                        car.setName(name);
```

```
                        car.setYear(Integer.parseInt(year));
                        car.setDistance(Integer.parseInt(distance));
                        car.setFuel(fuel);
                        car.setFuelConsumption(fuelConsumption);
                        car.setPrice(Integer.parseInt(price));
                    }
                    else {
                        System.out.println("Введите топливо: Бензин или
Дизель!");
                    }
                /* }
                 else {
                    System.out.println("Данные некорректны!");
                 }
                 if (companyService.updateCompany(company)) {
                    System.out.println("---Изменение выполнено!---");
                 }*/
            // }
        }
    }

    public void findCarByName() {
        System.out.print("Введите название машины: ");
        String name = in.nextLine();
        Car car = null;
        for (Company company : getCompanies()) {
            car = findCarInList(company, name);
            if (car != null) {
                theHeaderForCar();
                theTableForCar(car);
            }
        }
        System.out.println(" ");
    }
    private Car findCarInList(Company company, String name) {
        Car car = null;
        if (!company.getCars().isEmpty()) {
            for (Car c : company.getCars()) {
                if (c.getName().equals(name)) {
                    car = c;
                }
            }
            if (car == null) {
                System.out.println("Такой машины не найдено в компании " +
company.getCompanyName());
            }

        }
        return car;
    }
    public void showCarsFromOneCompany() {
        showCompanies();
        Company company = findCompanyByName();
        if (company != null) {
```

```
            if (!company.getCars().isEmpty()) {
                theHeaderForCar();
                for (Car c : company.getCars()) {
                    theTableForCar(c);
                }
                System.out.println(" ");
            }
            else {
                System.out.println("Компания " + company.getCompanyName() + " не
имеет моделей!");
            }
        }
    }

    public void showCars() {
        List<Company> companies = getCompanies();
        if (companies.size() != 0) {
            theHeaderForCar();
            for (Company c: companies) {
                List<Car> cars = c.getCars();
                if (!cars.isEmpty()) {
                    for (Car car : cars) {
                        theTableForCar(car);
                    }
                    System.out.println(" ");
                }

            }
        }
        else {
            System.out.println("Нет пользователей!");
        }
    }

    private void theHeaderForCar() {
        System.out.format("%15s%20s%10s%10s%10s%15s%10s%20s", "ID |", "Название
|", "Год |", "Пробег |", "Топливо |",
                "Расход топлива |", "Цена |", " Компания|");
    }

    private void theTableForCar(Car car) {
        System.out.println(" ");
        System.out.format("%15s%20s%10s%10s%10s%16s%10s%20s", car.getCarId() + "
|", car.getName() + " |",
                car.getYear() + " |", car.getDistance() + " |", car.getFuel() +
" |",
                car.getFuelConsumption() + " |", car.getPrice() + " |",
car.getCompany().getCompanyName() + " |");
    }
}
```

Вернемся к классу *LogIn* и рассмотрим часть логики для регистрации.

Из данного класса направляемся в следующий слой архитектуры приложения (рисунок 3.4).

Рисунок 3.4 – Уровень работы с бизнес-логикой

Данная часть отвечает за часть бизнес-логики, выполняемой системой. В частности, мы должны обработать работу с *Person*. Рассмотрим *PersonService*.

```
package service;
import entity.Person;
import java.util.List;

public interface PersonService {
    boolean addPerson(Person person);
    boolean updatePerson(Person person);
    boolean deletePerson(int id);
    List<Person> showPeople();
    Person findPersonById(int id);
}
```

Если классы *Menu*, *LogIn* и т. д. отвечают за порядок предоставления данных и связи пользовательского меню с обработкой действий системы, то начиная с уровня сервисов, мы описываем полноценную бизнес-логику приложения, как она работает с БД или иными источниками данных, как хранит информацию и какие действия с этой информацией может выполнить.

Наш интерфейс хранит методы, и мы их должны переопределить в *PersonServiceImpl*. Пока также мы рассматриваем часть – это первый метод.

```
package service.serviceImpl;
import dao.PersonDao;
import dao.daoImpl.PersonDaoImpl;
import entity.Person;
import exception.ShowException;
import org.hibernate.HibernateError;
import service.PersonService;
import java.util.List;

public class PersonServiceImpl implements PersonService {
```

```
PersonDao personDao = new PersonDaoImpl();

public PersonServiceImpl() {}

@Override
public boolean addPerson(Person person) {
    boolean isAdded = false;
    try {
        if (personDao.addPerson(person))
            isAdded = true;
    }
    catch (HibernateError e) {
        ShowException.showNotice(e);
    }
    return isAdded;
}
@Override
public boolean updatePerson(Person person) {
    boolean isUpdated = false;
    try {
        if (personDao.updatePerson(person))
            isUpdated = true;
    }
    catch (HibernateError e) {
        ShowException.showNotice(e);
    }
    return isUpdated;
}
@Override
public boolean deletePerson(int id) {
    boolean isDeleted = false;
    try {
        if (personDao.deletePerson(id))
            isDeleted = true;
    }
    catch (HibernateError e) {
        ShowException.showNotice(e);
    }
    return isDeleted;
}
@Override
public List<Person> showPeople() {
    List<Person> people = null;
    try {
        people = personDao.showPeople();
    }
    catch (HibernateError e) {
        ShowException.showNotice(e);
    }
    return people;
}
```

```
    @Override
    public Person findPersonById(int id) {
        Person person = null;
        try {
            person = personDao.findPersonById(id);
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return person;
    }
}
```

Разберем данный класс.

Из этого следует, что мы переходим еще глубже в логику – в слой работы с БД или, иными словами, *DAO* (*data access object*). Теперь становится ясным понятие «Объектный язык программирования» – начиная с простой инкапсуляции, *POJO* и уровнями архитектуры все завязано на этом представлении объектов и их трансформации (рисунок 3.5).

Начнем с *Entity*.

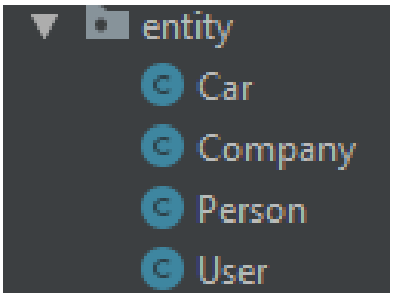


Рисунок 3.5 – *Entity*-классы приложения

А именно с *Person*:

```
package entity;
import javax.persistence.*;
@Entity
@Table(name = "people")
public class Person {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    @Column (name = "person_id")
    private int personId;
    @Column
    private String surname;
    @Column
    private String name;
```

```
    @Column
    private int age;
    @Column
    private String phone;
    @Column
    private String mail;
    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval =
true)
    private User user;
/Конструкторы getters@setters
    }
}
```

И после этого перейдем уже в *DAO* (рисунок 3.6).

Рисунок 3.6 – Уровень работы с базой данных

В интерфейсе *PersonDao*:

```
package dao;
import entity.Person;
import entity.User;
import java.util.List;
public interface PersonDao {
    boolean addPerson(Person person);
    boolean updatePerson(Person person);
    boolean deletePerson(int id);
    List<Person> showPeople();
    Person findPersonById(int id);
}
```

И реализации *Impl*:

```
package dao.daoImpl;
import dao.PersonDao;
import entity.Person;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import sessionFactory.SessionFactoryImpl;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import java.util.List;
```

```
public class PersonDaoImpl implements PersonDao {
    @Override
    public boolean addPerson(Person person) {
        boolean isAdded = false;
        try {
            Session session =
SessionFactoryImpl.getSessionFactory().openSession();
            Transaction tx = session.beginTransaction();
            session.save(person);
            tx.commit();
            session.close();
            isAdded = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isAdded;
    }
    @Override
    public boolean updatePerson(Person person) {
        boolean isUpdated = false;
        try {
            Session session =
SessionFactoryImpl.getSessionFactory().openSession();
            Transaction tx = session.beginTransaction();
            session.update(person);
            tx.commit();
            session.close();
            isUpdated = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isUpdated;
    }
    @Override
    public boolean deletePerson(int id) {
        boolean isDeleted = false;
        try {
            Session session =
SessionFactoryImpl.getSessionFactory().openSession();
            Person person = session.load(Person.class, id);
            Transaction tx = session.beginTransaction();
            session.delete(person);
            tx.commit();
            session.close();
            isDeleted = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isDeleted;
    }
```

```
    @Override
    public List<Person> showPeople() {
        List<Person> people =
(List<Person>)SessionFactoryImpl.getSessionFactory().openSession().createQuery("
FROM Person").list();
        return people;
    }
    @Override
    public Person findPersonById(int id) {
        Person person = null;
        try {
            Session session =
SessionFactoryImpl.getSessionFactory().openSession();
            Transaction tx = session.beginTransaction();
            CriteriaBuilder cb = session.getCriteriaBuilder();
            CriteriaQuery<Person> cr = cb.createQuery(Person.class);
            Root<Person> root = cr.from(Person.class);
            cr.select(root).where(cb.equal(root.get("personId"), id));
            person = session.createQuery(cr).getSingleResult();
            tx.commit();
            session.close();
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return person;
    }
}
```

И наконец, последний класс (рисунок 3.7).

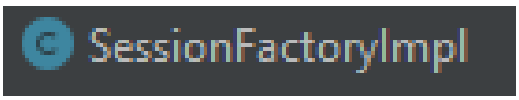


Рисунок 3.7 – Уровень настройки *Hibernate* в реализации приложения

```
package sessionFactory;
import entity.Car;
import entity.Company;
import entity.Person;
import entity.User;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
public class SessionFactoryImpl {
    private static SessionFactory sessionFactory;
    private SessionFactoryImpl() {}
    public static SessionFactory getSessionFactory() {
        if (sessionFactory == null) {
            try {
```

```
                Configuration configuration = new Configuration().configure();
                configuration.addAnnotatedClass(Company.class);
                configuration.addAnnotatedClass(Car.class);
                configuration.addAnnotatedClass(Person.class);
                configuration.addAnnotatedClass(User.class);
                StandardServiceRegistryBuilder builder = new
StandardServiceRegistryBuilder().applySettings(configuration.getProperties());
                sessionFactory =
configuration.buildSessionFactory(builder.build());
            } catch (Exception e) {
                System.out.println("Исключение!" + e);
            }
        }
        return sessionFactory;
    }
}
```

3.2 Структура БД

```
DROP SCHEMA IF EXISTS carshop;
CREATE SCHEMA carshop;
USE carshop;
 CREATE TABLE `companies` (
   `company_id` int NOT NULL AUTO_INCREMENT primary key,
   `company_name` varchar(45) NOT NULL,
   `company_country` varchar(45) NOT NULL);

 CREATE TABLE `cars` (
   `car_id` int NOT NULL AUTO_INCREMENT primary key,
   `name` varchar(45) NOT NULL,
   `year` int NOT NULL,
   `distance` int DEFAULT NULL,
   `fuel` varchar(45) NOT NULL DEFAULT 'Бензин',
   `fuel_consumption` varchar(45) NOT NULL,
   `price` int NOT NULL,
   `company_id` int NOT NULL,
   FOREIGN KEY (`company_id`) REFERENCES companies(`company_id`)
   ON delete cascade
 );
CREATE TABLE `people` (
   `person_id` int NOT NULL primary key auto_increment,
   `surname` varchar(45) NOT NULL,
   `name` varchar(45) NOT NULL,
   `age` int NOT NULL,
   `phone` varchar(45) NOT NULL,
   `mail` varchar(45) NOT NULL
);
CREATE TABLE `users` (
   `user_id` int NOT NULL primary key auto_increment,
   `login` varchar(45) NOT NULL,
   `password` varchar(45) NOT NULL,
   `role` varchar(45) not null DEFAULT 'User',
   `person_id` int,

 constraint `fk_user_person` foreign key (`person_id`) references `people`
(`person_id`)
 on delete cascade
);
```

3.3 Создание *Entity*-классов для маппинга объектной и реляционной модели

Рассмотрим БД и добавим *Entity* в проект.

Класс *Car*:

```
package entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
@Entity
@Table(name = "cars")
public class Car {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    @Column(name = "car_id")
    private int carId;
    @Column(name = "name")
    private String name;
    @Column(name = "year")
    private int year;
    @Column(name = "distance")
    private int distance;
    @Column(name = "fuel")
    private String fuel;
    @Column(name = "fuel_consumption")
    private String fuelConsumption;
    @Column(name = "price")
    private int price;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "company_id")
    private Company company;
/Конструкторы getters@setters@toString
    }
}
```

Класс *Company*:

```
package entity;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;
@Entity
@Table (name = "companies")
public class Company {
```

```
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    @Column(name = "company_id")
    private int companyId;
    @Column(name = "company_name")
    private String companyName;
    @Column(name = "company_country")
    private String companyCountry;
    @OneToMany(mappedBy = "company", cascade = CascadeType.ALL, orphanRemoval =
true, fetch = FetchType.EAGER)
    private List<Car> cars;
/Конструкторы getters@setters@toString
    }
}
```

Класс *User*:

```
package entity;
import javax.persistence.*;
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private int userId;
    @Column
    private String login;
    @Column
    private String password;
    @Column
    private String role;
    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;
/Конструкторы getters@setters
    }
}
```

Создадим оставшиеся интерфейсы и классы. В качестве тренировки реализуйте недостающие методы в классах сервис-слоя и работы с БД.

CompanyService:

```
package service;
import entity.Company;
import java.util.List;
public interface CompanyService {
    boolean addCompany(Company company);
    boolean updateCompany(Company company);
    boolean deleteCompany(int id);
    List<Company> showCompanies();
    Company findCompanyById(int id);
    Company findCompanyByName(String name);
}
```

CompanyDao:

```
package dao;
import entity.Company;
import java.util.List;

public interface CompanyDao {
    boolean addCompany(Company company);
    boolean updateCompany(Company company);
    boolean deleteCompany(int id);
    List<Company> showCompanies();
    Company findCompanyById(int id);
    Company findCompanyByName(String name);
}
```

ServiceCompanyImpl:

```
package service.serviceImpl;
import dao.CompanyDao;
import dao.daoImpl.CompanyDaoImpl;
import entity.Company;
import exception.ShowException;
import org.hibernate.HibernateError;
import service.CompanyService;
import java.util.List;
public class CompanyServiceImpl implements CompanyService {
    CompanyDao companyDao = new CompanyDaoImpl();
    public CompanyServiceImpl() {}
    @Override
    public boolean addCompany(Company company) {
        boolean isAdded = false;
        try {
            companyDao.addCompany(company);
            isAdded = true;
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return isAdded;
    }
    @Override
    public boolean updateCompany(Company company) {
        boolean isUpdated = false;
        try {
            companyDao.updateCompany(company);
            isUpdated = true;
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return isUpdated;
    }
```

```
    @Override
    public boolean deleteCompany(int id) {
        boolean isDeleted = false;
        try {
            companyDao.deleteCompany(id);
            isDeleted = true;
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return isDeleted;
    }
    @Override
    public List<Company> showCompanies() {
        List<Company> companies = null;
        try {
            companies = companyDao.showCompanies();
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return companies;
    }
    @Override
    public Company findCompanyById(int id) {
        Company company = null;
        try {
            company = companyDao.findCompanyById(id);
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return company;
    }
    @Override
    public Company findCompanyByName(String name) {
        Company company = null;
        try {
            company = companyDao.findCompanyByName(name);
        }
        catch (HibernateError e) {
            ShowException.showNotice(e);
        }
        return company;
    }
}
```

CompanyDaoImpl:

```
package dao.daoImpl;
import dao.CompanyDao;
import entity.Company;
import java.util.List;
```

```
public class CompanyDaoImpl implements CompanyDao {
    public CompanyDaoImpl() {
    }
    @Override
    public boolean addCompany(Company company) {
        boolean isAdded = false;
        try {
            Session                         session                         =
SessionFactoryImpl.getSessionFactory().openSession();
            Transaction tx = session.beginTransaction();
            session.save(company);
            tx.commit();
            session.close();
            isAdded = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isAdded;
    }
    @Override
    public boolean updateCompany(Company company) {
        boolean isUpdated = false;
        try {
            Session                         session                         =
SessionFactoryImpl.getSessionFactory().openSession();
            Transaction tx = session.beginTransaction();
            session.update(company);
            tx.commit();
            session.close();
            isUpdated = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isUpdated;
    }
    @Override
    public boolean deleteCompany(int id) {
        boolean isDeleted = false;
        try {
            Session                         session                         =
SessionFactoryImpl.getSessionFactory().openSession();
            Company company = session.load(Company.class, id);
            Transaction tx = session.beginTransaction();
            session.delete(company);
            tx.commit();
            session.close();
            isDeleted = true;
        }
        catch (NoClassDefFoundError e) {
            System.out.println("Exception: " + e);
        }
        return isDeleted;    }
```

```
        @Override
        public Company findCompanyById(int id) {
            Company company = null;
            try {
                Session                         session                         =
SessionFactoryImpl.getSessionFactory().openSession();
                Transaction tx = session.beginTransaction();
                CriteriaBuilder cb = session.getCriteriaBuilder();
                CriteriaQuery<Company> cr = cb.createQuery(Company.class);
                Root<Company> root = cr.from(Company.class);
                cr.select(root).where(cb.equal(root.get("companyId"), id));
                company = session.createQuery(cr).getSingleResult();
                tx.commit();
                session.close();
            }
            catch (NoClassDefFoundError e) {
                System.out.println("Exception: " + e);
            }
            return company;
        }
        @Override
        public Company findCompanyByName(String name) {
            Company company = null;
            try {
                Session                         session                         =
SessionFactoryImpl.getSessionFactory().openSession();
                Transaction tx = session.beginTransaction();
                CriteriaBuilder cb = session.getCriteriaBuilder();
                CriteriaQuery<Company> cr = cb.createQuery(Company.class);
                Root<Company> root = cr.from(Company.class);
                cr.select(root).where(cb.equal(root.get("companyName"),
name));
                company = session.createQuery(cr).getSingleResult();
                tx.commit();
                session.close();
            }
            catch (NoClassDefFoundError e) {
                System.out.println("Exception: " + e);
            }
            return company;
        }
        @Override
        public List<Company> showCompanies() {
            List<Company>                       companies                       =
(List<Company>)SessionFactoryImpl.getSessionFactory().openSession().createQuery(
"From Company").list();
            return companies;
        }
    }
```

Создайте класс *Validator*:

```
package validator;
import java.util.regex.Pattern;

public class Validator {

    private static final Pattern STRING_PATTERN =
Pattern.compile("^[\\p{L}]+$");
    private static final Pattern NUMBER_PATTERN = Pattern.compile("\\d+");
    private static final Pattern FUEL_CONSUMPTION_PATTERN = Pattern.compile("\\-
?\\d+(\\.\\d{0,})?");
    public static boolean correctCompany(String name, String country) {
        boolean isCorrect = true;
        if (name.isBlank() || country.isBlank() ||
!STRING_PATTERN.matcher(country).matches()) {
            isCorrect = false;
        }
        return  isCorrect;
    }
    public static boolean correctPerson(String name, String surname, String age,
String phone, String mail) {
        boolean isCorrect = true;
        if (name.isBlank() || surname.isBlank() || age.isBlank() ||
!NUMBER_PATTERN.matcher(age).matches() ||
                phone.isBlank() || mail.isBlank()) {
            isCorrect = false;
        }
        return isCorrect;
    }
    public static boolean correctUser(String login, String password) {
        boolean isCorrect = true;
        if (login.isBlank() || password.isBlank()) {
            isCorrect = false;
        }
        return isCorrect;
    }
    public static boolean correctRole(String role) {
        boolean isCorrect = true;
        if (!role.equals("User") || !role.equals("Admin")) {
            isCorrect = false;
        }
        return isCorrect;
    }
    public static boolean correctId(String id) {
        boolean isCorrect = true;
        if (id.isBlank() || !NUMBER_PATTERN.matcher(id).matches()) {
            isCorrect = false;
        }
        return isCorrect;
    }
```

```
    public static boolean correctCar(String name, String year, String distance,
                                     String fuel, String fuelConsumption, String
price) {
        boolean isCorrect = true;
        if (name.isBlank() || year.isBlank() ||
!NUMBER_PATTERN.matcher(year).matches() || distance.isBlank() ||
                !NUMBER_PATTERN.matcher(distance).matches() || fuel.isBlank() ||
fuelConsumption.isBlank() ||
                !FUEL_CONSUMPTION_PATTERN.matcher(fuelConsumption).matches() ||
price.isBlank() ||
                !NUMBER_PATTERN.matcher(price).matches()) {
            isCorrect = false;
        }
        return isCorrect;
    }
    public static boolean correctFuel(String fuel) {
        boolean isCorrect = false;
        if (fuel.equals("Бензин") || fuel.equals("Дизель")) {
            isCorrect = true;
        }
        return isCorrect;
    }
}
```

Тема 4. Реализация веб-приложения на основе *Servlets*

4.1 Конфигурация проекта

Создадим сервлет в среде разработки *Idea*. В зависимости от версии форма окон может немного изменяться. Необходимо выбрать сборщик проекта, в данном случае *Maven*, а также пункт *Servlets*.

В вашем варианте будет примерно так, как показано на рисунках 4.1 и 4.2.

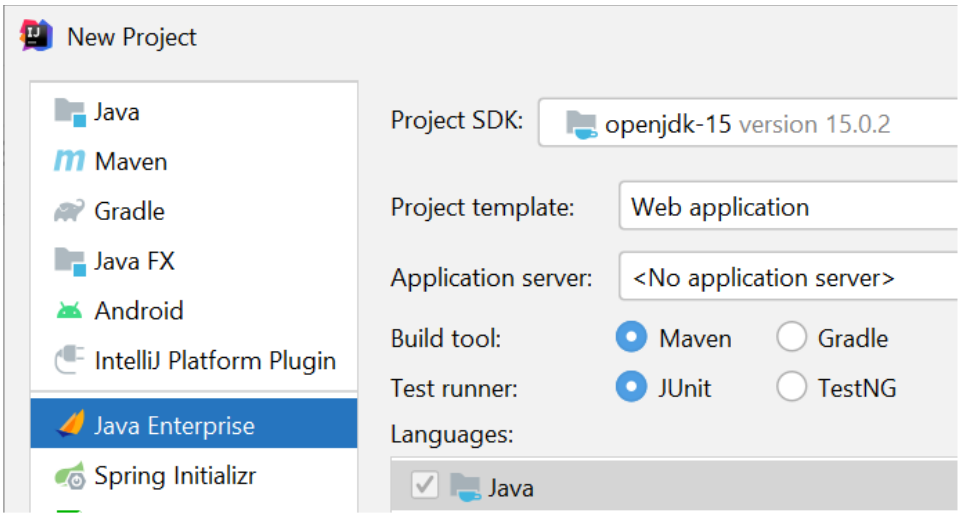


Рисунок 4.1 – Инициализация проекта

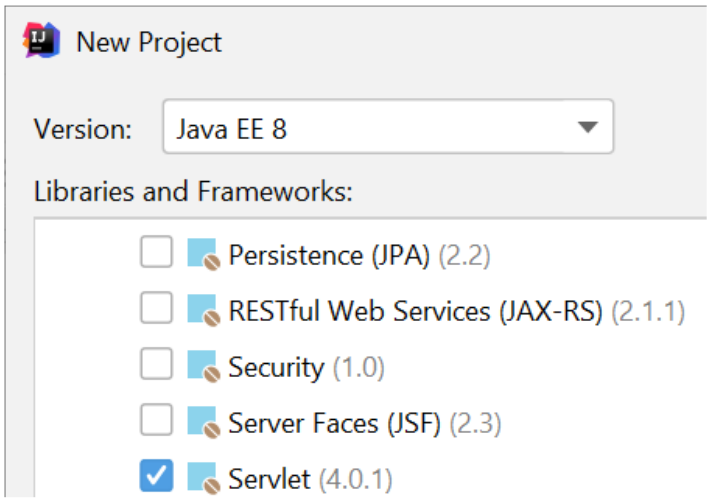


Рисунок 4.2 – Выбор библиотек

Также вам будут предложены варианты версий *Tomcat*.

В старой версии *Idea* подключение можно выполнить таким способом (рисунок 4.3).

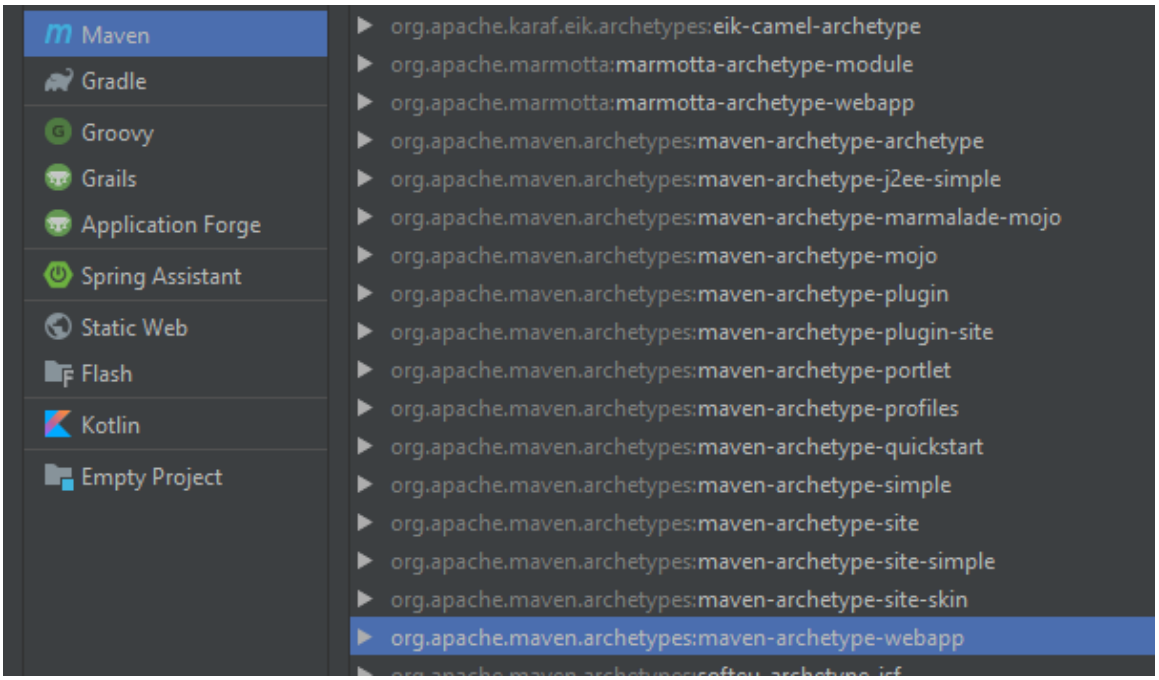


Рисунок 4.3 – Подключение библиотек через архетип в старой версии *Idea*

Структура проекта будет следующая (рисунок 4.4).

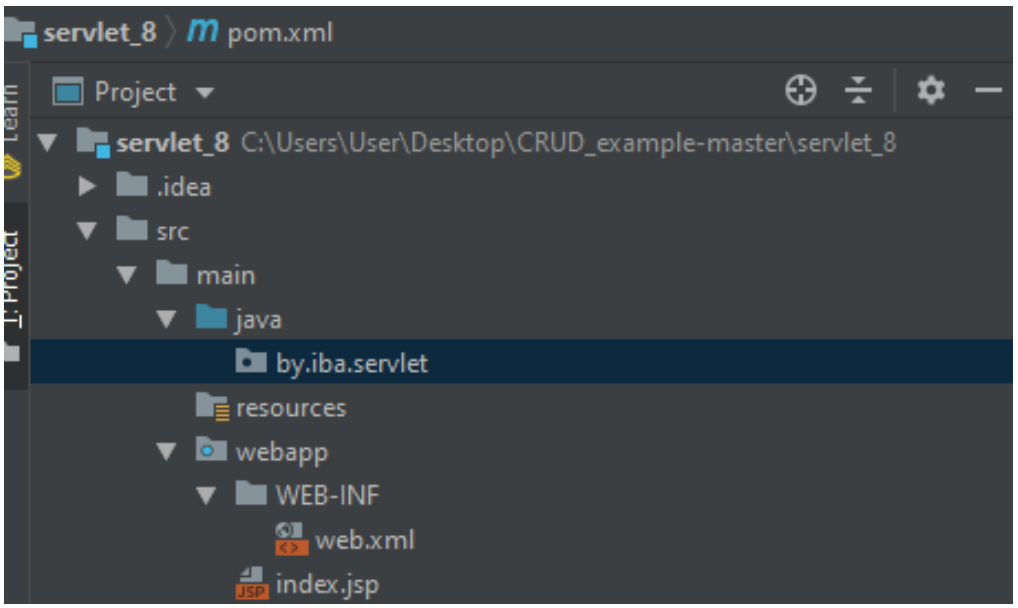


Рисунок 4.4 – Структура проекта

Если вы используете *Tomcat* версии 10, то надо будет вместо *javax.servlet* использовать *jakarta.servlet* и в *pom.xml* установить зависимость

```
<dependency>
	<groupId>jakarta.servlet</groupId>
	<artifactId>jakarta.servlet-api</artifactId>
	<version>5.0.0</version>
	<scope>provided</scope>
</dependency>
```

Добавим еще один *New →Servlet***.** Назовем его *LoginServlet* (рисунок 4.5).

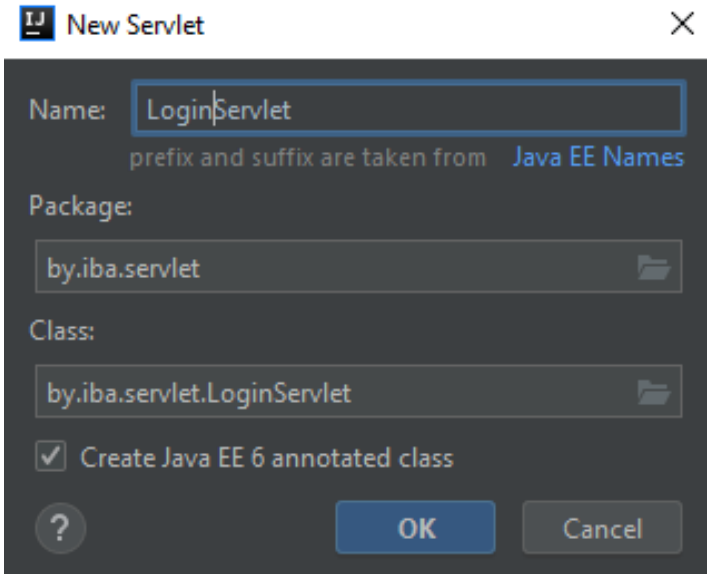


Рисунок 4.5 – Добавление сервлета

Чтобы создать сервлет, нам нужен суперкласс *HttpServlet*, который определен в *Java EE Web API*. Нужные зависимости для поддержки сервлетов уже есть в проекте (рисунок 4.6).

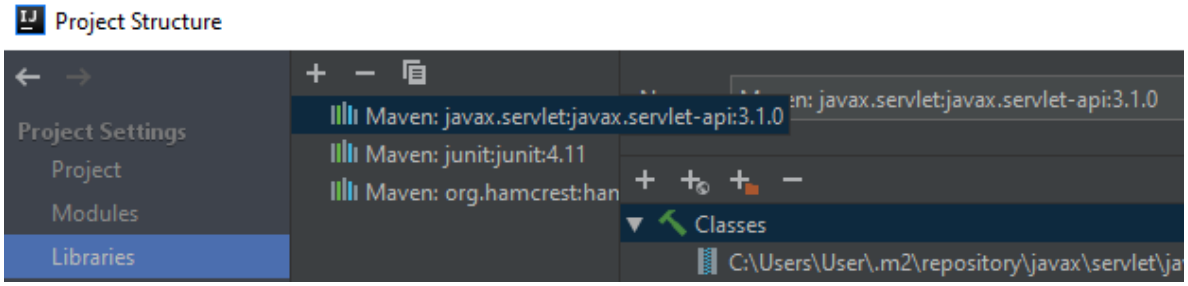


Рисунок 4.6 – Проверка зависимостей

Посмотрим на добавленный класс сервлета. У вас уже есть два автоматически сгенерированных метода:

```
@WebServlet(name = "LoginServlet")
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    }
}
```

4.2 Жизненный цикл *Servlets*

Сервлеты выполняются на платформе веб-сервера как часть того же процесса, что и сам веб-сервер. Веб-сервер отвечает за инициализацию, вызов и уничтожение каждого экземпляра сервлета. Веб-сервер взаимодействует с сервлетом через простой интерфейс: javax.servlet.Servlet.

Интерфейс javax.servlet.Servlet состоит из трех главных методов:

1) *init()*;
2) *service()*;
3) *destroy()*.

И двух вспомогательных методов:

1) *getServletConfig()*;
2) *getServletInfo()*.

Сходство между интерфейсами сервлета и апплета *Java* очевидны. Именно так и было спроектировано. Сервлеты являются для веб-серверов тем же самым,

чем являются апплеты для веб-браузеров. Апплет выполняется в веб-браузере, совершая действия по его запросу через специальный интерфейс. Сервлет делает то же самое, работая на веб-сервере.

Метод *init()*

При первой загрузке сервлета вызывается метод *init()*. Это дает возможность сервлету выполнить любую работу по установке, например, открытие файлов или установку соединений с их серверами. Если сервлет установлен на сервере постоянно, он загружается при запуске сервера. В противном случае сервер активизирует сервлет при получении первого запроса от клиента на выполнение услуги, обеспечиваемой этим сервлетом. Гарантируется, что метод *init()* закончится перед любым другим обращением к сервлету, таким как, например, вызов метода *service()*. Обратите внимание, что *init()* будет вызван только один раз; он не будет вызываться до тех пор, пока сервлет не будет выгружен и затем загружен сервером снова. Метод *init()* принимает один аргумент – ссылку на объект *ServletConfig*, который содержит аргументы для инициализации сервлета. Этот объект имеет метод *getServletContext()*, возвращающий объект *ServletContext*, который содержит информацию об окружении сервлета.

Метод *service()*

Метод *service()* является сердцем сервлета. Каждый запрос от клиента приводит к одному вызову метода *service()*. Этот метод читает запрос и формирует ответное сообщение при помощи своих двух аргументов *ServletRequest* и *ServletResponse*:

1) Объект *ServletRequest* содержит данные от клиента. Данные состоят из пар «имя:значение» и *InputStream*. Существует несколько методов, возвращающих информацию о параметрах клиента. *InputStream* может быть получен при помощи метода *getInputStream()*. Этот метод возвращает объект *ServletInputStream,* который можно использовать для получения дополнительных данных от клиента. Если необходимо выполнить обработку символьных данных, а не двоичных, то можно получить объект *BufferedReader* при помощи метода *getReader()*.

2) Объект *ServletResponse* содержит ответ сервлета клиенту. Во время подготовки ответа прежде всего вызывается метод *setContentType()* для установки типа *MIME*-ответа. Затем могут быть использованы методы *getOutputStream()* или *getWriter()* для получения объектов и *ServletOutputStream* или *PrintWriter* соответственно для передачи данных обратно клиенту.

Таким образом, существуют два способа передачи информации от клиента к сервлету. Первый – через передачу значений в параметрах запроса. Значения

параметров могут быть вставлены в *URL*. Второй способ передачи информации от клиента к сервлету осуществляется через *InputStream* (или *Reader*).

Работа метода *service()* по существу проста – он создает ответ на каждый клиентский запрос, переданный ему с сервера. Однако необходимо помнить, что могут существовать несколько параллельных запросов, обрабатываемых в одно и то же время. Если метод *service()* требует каких-либо внешних ресурсов, таких как файлы, базы данных, то необходимо гарантировать, чтобы доступ к ресурсам являлся потокозащищенным.

Метод *destroy()*

Метод *destroy()* вызывается для освобождения всех ресурсов (например, открытые файлы и соединения с базой данных) перед выгрузкой сервлета. Этот метод может быть пустым, если нет необходимости выполнения каких-либо завершающих операций. Перед вызовом метода *destroy()* сервер ждет либо завершения всех обслуживающих операций, либо истечения определенного времени. Это означает, что метод *destroy()* может быть вызван во время выполнения какого-либо продолжительного метода *service()*. Важно оформить метод *destroy()* таким образом, чтобы избежать закрытия необходимых ресурсов до тех пор, пока все вызовы *service()* не завершатся.

Метод *getServletConfig()*

Метод *getServletConfig()* возвращает ссылку на объект, который реализует интерфейс *ServletConfig*. Данный объект предоставляет доступ к информации о конфигурации сервлета, т. е. доступ к параметрам инициализации сервлета и объекту контекста сервлета *ServletContext*, который дает доступ к сервлету и его окружению.

Метод *getServletInfo()*

Метод *getServletInfo()* определяется программистом, создающим сервлет, для возврата строки, содержащей информацию о сервлете, например: автор и версия сервлета.

Так, когда пользователь отправляет запрос *Servlet*, который создается в момент получения первого запроса, одновременно вызывается метод *init()* в *Servlet*, чтобы инициализировать его, *init()* вызывается один-единственный раз. Метод *destroy()* используется для разрушения *Servlet*, вызывается единственный раз когда вы отменяете развертывание (*undeloy*) веб-приложения или отключаете (shutdown) *Web Server* (веб-сервер).

Поскольку для обработки всех запросов создается один экземпляр сервлета и все обращения к нему идут в отдельных потоках, то не рекомендуется в классе сервлета объявлять и использовать глобальные переменные, так как они не будут потокобезопасными.

Когда пользователь вводит ссылку в браузере, то она будет отправлена в *WebContainer*. *WebContainer* должен решить, какой *Servlet* будет обслуживать этот запрос от пользователя. Для этого ему надо определить *urlPatterns*.

Servlet с *urlPatterns* = "/ "

Это *Servlet* по умолчанию и он будет использоваться для обработки запроса (*request*), который имеет ссылку, не совпадающую ни с каким *urlPatterns* другого *Servlet,* объявленного в вашем приложении.

Начиная с версии 3.0 можно конфигурировать *Servlet,* используя *Annotation*.

Поменяйте код на следующий:

```
package by.iba.servlet;
import javax.servlet.annotation.WebServlet;

@WebServlet(name = "LoginServlet", urlPatterns = "/LoginServlet")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Good morning </title>");
        out.println("</head>");
        out.println("<body>");
        out.println(" First Servlet");
        out.println("</body>");
        out.println("</html>");
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    }}
```

Перед определением класса указана аннотация *WebServlet*, которая уточняет, с какой конечной точкой будет сопоставляться данный сервлет, т. е. данный сервлет будет обрабатывать запросы по адресу "/*LoginServlet*".

Для обработки *GET*-запросов (например, при обращении к сервлету из адресной строки браузера) сервлет должен переопределить метод *doGet*. В данном случае *GET*-запрос по адресу "*/LoginServlet*" будет обрабатываться методом *doGet*.

Этот метод принимает два параметра.

Параметр типа *HttpServletRequest* инкапсулирует всю информацию о запросе, а параметр типа *HttpServletResponse* позволяет управлять ответом. С помощью метода *getWriter*() объекта *HttpServletResponse* мы можем получить

объект *PrintWriter*, через который можно отправить какой-то определенный ответ пользователю.

В данном случае через метод *println()* пользователю отправляется простейший *html*-код. По завершении использования объекта *HttpServletResponse* его необходимо закрыть с помощью метода *close()*.

Зайдите в файл *WEB-INF\web.xml* и внесите изменения. Так как мы используем аннотации, то настроек в дескрипторе развертывания нет.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
    <welcome-file-list>
        <welcome-file>LoginServlet</welcome-file>
    </welcome-file-list>
</web-app>
```

Мы настроили *welcome*-файл логина.

Запустите проект (рисунок 4.7).

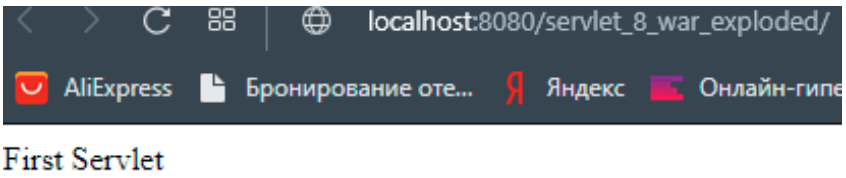


Рисунок 4.7 – Первый запуск приложения с *GET*-запросом

Мы определили метод *doGet* (запрос *HttpServletRequest*, ответ *HttpServletResponse*) в *LoginServlet*. Метод *doGet* обрабатывает запросы *GET* к шаблону *URL* "/*LoginServlet*".

Переопределите еще три метода жизненного цикла: *init*, *destroy*, *service*.

Итак, для обработки запроса в *HttpServlet* определен ряд методов, которые мы можем переопределить в классе сервлета:

- *doGet*: обрабатывает запросы *GET* (получение данных);
- *doPost*: обрабатывает запросы *POST* (отправка данных);
- *doPut*: обрабатывает запросы *PUT* (отправка данных для изменения);
- *doDelete*: обрабатывает запросы *DELETE* (удаление данных);
- *doHead*: обрабатывает запросы *HEAD*.

Каждый метод обрабатывает определенный тип запросов *HTTP* и мы можем определить все эти методы, но зачастую работа идет в основном с методами *doGet* и *doPost*.

Когда вы вводите *URL*-адрес браузера, браузер создает *HTTP*-запрос *GET*. Запрос *HTTP GET* получен веб-приложением, развернутым на сервере *Tomcat*.

Сервлеты. Контекст.

Сервлет может получать информацию о своем окружении в различное время. Во время запуска сервлета доступна информация об инициализации; информация о сервере доступна в любое время. Кроме этого, любой запрос может содержать дополнительную специфическую информацию.

Информация об инициализации сервера

Информация об инициализации сервера передается сервлету при помощи параметра *ServletConfig* метода *init()*. Каждый веб-сервер обеспечивает свой способ передачи информации об инициализации в сервлет. Если, например, класс сервера *DatePrintServlet* принимает аргумент инициализации *timezone*, то необходимо определить следующие свойства в файле *servlets.properties*:

– *servlet.dateprinter.code=DatePrinterServlet;*
– *servlet.dateprinter.timezone=PST*.

Эта информация также может быть предоставлена административным средством *GUI*. В следующем коде сервлет получает доступ к информации *timezone* на этапе инициализации.

Информация о контексте сервера

Информация о контексте сервера доступна в любое время через объект *ServletContext*. Сервлет может получить этот объект, вызывая метод *getServletContext()* объекта *ServletConfig*. Необходимо помнить, что этот объект передается сервлету во время инициализации. Грамотно написанный метод *init()* сохраняет ссылку в переменной, имеющей тип доступа *private*. Интерфейс *ServletContext* определяет несколько методов, представленных в таблице 4.1.

Таблица 4.1 – Описание методов контекста сервлета

Метод	Описание
getAttribute ()	Гибкий способ получения информации о сервере через пары атрибутов «имя:значение»
getMimeType ()	Возвращает тип *MIME* данного файла
getRealPath ()	Этот метод преобразует относительный или виртуальный путь в новый путь относительно месторасположения корня *HTML*-документов сервера

Метод	Описание
getServerInfo ()	Возвращает имя и версию сетевой службы, в которой исполняется сервлет
getServlet ()	Возвращает объект *Servlet* указанного имени. Полезен при доступе к службам других сервлетов
getServletNames ()	Возвращает список имен сервлетов, доступных в текущем пространстве имен
log ()	Записывает информацию в файл регистрации сервлета. Имя файла регистрации и его формат зависят от сервера

Следующий пример показывает, как сервлет использует веб-сервер для записи сообщения в свой *log*-файл во время инициализации:

```
public HelloWorld implements Servlet
{
private ServletConfig config;
public void init (ServletConfig config) throws ServletException
{
this.config = config;
ServletContext sc = config.getServletContext();
sc.log( "Started OK!" );
} }
```

Контекст сервлета во время запроса на обслуживание

Каждый запрос на обслуживание может содержать информацию в форме пар параметров «имя:значение», как *ServletInputStream* или *BufferedReader*. Эта информация доступна при помощи объекта *ServletRequest*, который передается в метод *service()*. Следующий код показывает, как получить информацию во время работы метода *service()*:

```
BufferedReader reader;
String param1;
String param2;
public void service(ServletRequest request,ServletResponse response)
{
reader = request.getReader();
param1 = request.getParameter("First");
param2 = request.getParameter("Second");
}
```

Существует также дополнительная информация, доступная сервлету через объект *ServletRequest*. Она приведена в таблице 4.2.

Таблица 4.2 – Описание методов запроса на обслуживание сервлета

Метод	Описание
getAttribute ()	Возвращает значение указанного атрибута этого запроса
getContentLength ()	Размер запроса, если известен
getContentType ()	Возвращает тип *MIME* тела запроса
getInputStream ()	Возвращает *InputStream* для чтения двоичных данных из тела запроса
GetParameterNames ()	Возвращает массив строк с именами всех параметров
getParameterValues ()	Возвращает массив значений для указанного параметра
getProtocol ()	Возвращает протокол и версию для запроса как строку вида /..
getReader ()	Возвращает *BufferedReader* для получения текста из тела запроса
getRealPath ()	Возвращает реальный путь для указанного виртуального пути
getRemoteAddr ()	*IP*-адрес клиента, пославшего данный запрос
getRemoteHost ()	Имя хоста клиентской машины, пославшего данный запрос
getScheme ()	Возвращает схему, используемую в *URL* этого запроса (например, *https, http, ftp* и т. д.)
getServerName ()	Имя хоста сервера, принявшего данный запрос
getServerPort ()	Возвращает номер порта, используемого для приема этого запроса

Интерфейс *HttpServletRequest*

При каждом вызове методы *doGet* и *doPost* класса *HttpServlet* принимают в качестве параметра объект, который реализует интерфейс *HttpServletRequest*. Веб-сервер, который исполняет сервлет, создает объект *HttpServletRequest* и передает его методу *service* сервлета, который в свою очередь передает его методу *doGet* или *doPost*. Данный объект содержит запрос, поступивший от клиента.

Имеется множество методов (таблица 4.3), дающих возможность сервлету обрабатывать клиентский запрос. Некоторые из этих методов принадлежат интерфейсу *ServletRequest*, который расширяется интерфейсом *HttpServletRequest*. Ряд ключевых методов, использованных в примерах, представлен в таблице 4.3. Полный список методов интерфейса *HttpServletRequest* можно найти в документации компании *Sun*.

Таблица 4.3 – Описание методов обработки запросов сервлета

Метод	Описание
String getParameter(String name)	Получение из запроса значения параметра. Наименование параметра определено значением *name*
Enumeration getParameterNames()	Получение из запроса имен всех параметров
String[] getParameterValues (String name)	Для параметра с несколькими значениями данный метод возвращает строковый массив
Cookie[] getCookies ()	Получение массива объектов *Cookie*, сохраненных на компьютере клиента
HttpSession getSession(boolean create)	Возвращает объект *HttpSession* текущего сеанса клиента

Интерфейс *HttpServletResponse*

При каждом обращении к сервлету методы *doGet* и *doPost* класса *HttpServlet* принимают объект, который реализует интерфейс *HttpServletResponse*. Веб-сервер, который исполняет сервлет, создает объект *HttpServletResponse* и передает его методу *service* сервлета. Объект *HttpServletResponse* описывает ответ клиенту.

Имеется множество методов (таблица 4.4), дающих возможность сервлету сформировать ответ клиенту. Некоторые из этих методов принадлежат интерфейсу *ServletResponse*, расширяемому интерфейсом *HttpServletResponse*.

Таблица 4.4 – Описание методов ответа сервлета

Метод	Описание
void addCookie (Cookie cookie)	Метод используется для добавления *Cookie* в заголовок ответа клиенту. Установленный максимальный возвраст *Cookie*, а также разрешение клиентом хранения *Cookie* определяют, будут ли *Cookies* сохранены на клиенте и время их хранения
ServletOutputStream getOutputStream()	Получение бинарного потока вывода для отправления бинарных данных клиенту
PrintWriter getWriter	Получение символьного потока вывода для отправления текстовых данных клиенту
void setContentType(String type)	Определение *MIME*-типа ответа браузеру. *MIME*-тип помогает браузеру определить, как отображать данные. Например, *MIME*-тип «*text/html*» указывает, что ответ является *HTML*-документом, поэтому браузер отображает *HTML*-страницу

4.3 Добавление *JSP*

Сервлеты позволяют получать запросы от клиента, совершать некоторую работу и выводить результаты на экран. Сервлет работает до того момента, пока речь идет об обработке информации, т. е. до вывода информации на экран. В сервлет можно вставить достаточно сложную логику, сделать вызовы к базе данных и многое другое, что необходимо для приложения. Но вот осуществлять вывод на экран внутри самого сервлета – очень неудобно.

Технология проектирования *Java Server Pages (JSP)* – это одна из технологий *JavaEE*, которая представляет собой расширение технологии сервлетов для упрощения работы с веб-содержимым. Страницы *JSP* позволяют легко разделить веб-содержимое на статическую и динамическую часть, допускающую многократное использование ранее определенных компонентов. Разработчики *Java Server Pages* могут использовать компоненты *JavaBeans* и создавать собственные библиотеки нестандартных тегов, которые инкапсулируют сложные динамические функциональные средства.

Во многих случаях сервлеты и *JSP*-страницы являются взаимозаменяемыми. Подобно сервлетам, *JSP*-страницы обычно выполняются на стороне веб-сервера, который называют контейнером *JSP*.

Когда веб-сервер, поддерживающий технологию *JSP*, принимает первый запрос на *JSP*-страницу, контейнер *JSP* транслирует эту *JSP*-страницу в сервлет *Java*, который обслуживает текущий запрос и все последующие запросы к этой странице. Если при компиляции нового сервлета возникают ошибки, эти ошибки приводят к ошибкам на этапе компиляции. Контейнер *JSP* на этапе трансляции помещает операторы *Java*, которые реализуют ответ *JSP*-страницы, в метод *jspService*. Если сервлет компилируется без ошибок, контейнер *JSP* вызывает метод *jspService* для обработки запроса.

JSP-страница может обработать запрос непосредственно либо вызвать другие компоненты веб-приложения, чтобы содействовать обработке запроса. Любые ошибки, которые возникают в процессе обработки, вызывают исключительную ситуацию в веб-сервере на этапе запроса.

Весь статический текст *HTML*, называемый в документации *JSP*-шаблоном *HTML (template HTML)*, сразу направляется в выходной поток. Выходной поток страницы буферизуется. Буферизацию обеспечивает класс *JspWriter*, расширяющий класс *Writer*.

Таким образом, достаточно написать страницу *JSP*, сохранить ее в файле с расширением *jsp* и установить файл в контейнер, так же как и страницу *HTML*, не заботясь о компиляции. При установке можно задать начальные параметры страницы *JSP* так же, как и начальные параметры сервлета.

Архитектура *JSP*-страницы

Страница *JSP* располагается на веб-сервере в среде виртуальной *Java*-машины. Доступ к страницам *JSP*, как и в случае сервлета, осуществляется через *Web* с использованием протокола *HTTP*.

Страница *JSP* функционирует под управлением *JSP Engine* (среды исполнения *JSP*). Страница *JSP* может взаимодействовать с программным окружением с помощью компонентов *JavaBeans*, получая и устанавливая его параметры, используя теги: *<jsp:useBean>, <jsp:getProperty>, <jsp:setProperty>*. Компонент *JavaBean* сам может участвовать в других процессах, предоставляя результаты в виде своих параметров, доступных страницам *JSP*, участвующим в сеансе, а через них – всем пользователям, запрашивающим эти страницы *JSP*.

Основные модели архитектуры *JSP*

Возможны различные подходы к использованию технологии *JSP*. Два основных архитектурных подхода, нашедшие применение при реализации приложений уровня предприятия, имеют специальные названия:

1) *JSP Model 1* (Первая модель архитектуры *JSP*);

2) *JSP Model 2* (Вторая модель архитектуры *JSP*).

Первая модель архитектуры *JSP*

*JSP Model 1 (*Первая модель архитектуры *JSP)* практически реализует базовую архитектуру *JSP*.

В архитектурном решении *JSP Model 1* полностью отвечает за получение запроса от клиента, его обработку, подготовку ответа и доставку ответа пользователю. Разделение представления и динамического содержания обеспечивается тем, что доступ к данным осуществляется через компоненты *JavaBeans*.

В сценарии *JSP Model 1* предполагается следующая последовательность действий (рисунок 4.8):

1) запрос пользователя посылается через веб-браузер странице *JSP*;

2) страница *JSP* компилируется в сервлет (при первом обращении) ;

3) скомпилированный сервлет обращается к некоторому компоненту *JavaBean*, запрашивая у него информацию;

4) компонент *JavaBean*, в свою очередь, осуществляет доступ к информационным ресурсам (непосредственно или через компонент *Enterprise JavaBeans*);

5) полученная информация отображается в свойствах компонента *JavaBeans*, доступных на странице *JSP*;

6) формируется ответ в виде страницы *HTML* с комбинированным содержанием (статическое, динамическое).

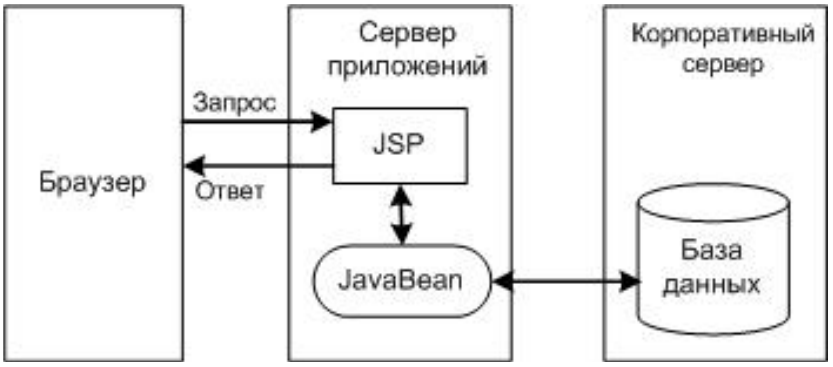


Рисунок 4.8 – Порядок *работы JSP Model 1*

Архитектура *JSP Model 1* может с успехом применяться для небольших приложений. Однако использование данной модели для более сложных задач вызывает определенные трудности и не является технологичным из-за большого объема встроенных в страницу программных фрагментов. Для сложных корпоративных приложений рекомендуется применение второй модели архитектуры *JSP*.

Вторая модель архитектуры *JSP*

*JSP Model 2 (*Вторая модель архитектуры *JSP)* реализует гибридный подход к обслуживанию динамического содержания веб-страницы, при котором совместно используется сервлет и страница *JSP*.

Эта модель позволяет эффективно использовать преимущества обеих технологий: сервлет поддерживает задачи, связанные с обработкой запроса и созданием объектов *JavaBeans*, используемых *JSP*, а страница *JSP* отвечает за визуальное представление информации. Сервлет используется как управляющее устройство (контроллер). Схематично вторая модель представлена на рисунке 4.9.

Сценарий *JSP Model 2*, как правило, реализует следующую типовую последовательность действий (рисунок 4.9):

1) запрос пользователя посылается через веб-браузер сервлету;

2) сервлет обрабатывает запрос, создает и инициализирует объект *JavaBean* или другие объекты, используемые страницей *JSP*, и запрашивает динамическое содержание у компонента *JavaBean*;

3) компонент *JavaBean* осуществляет доступ к информации непосредственно или через компонент *Enterprise JavaBeans*;

4) сервлет, направляющий запрос, вызывает сервлет, скомпилированный из страницы *JSP*;

5) сервлет, скомпилированный из страницы *JSP*, встраивает динамическое содержание в статический контекст *HTML*-страницы и отправляет ответ пользователю.

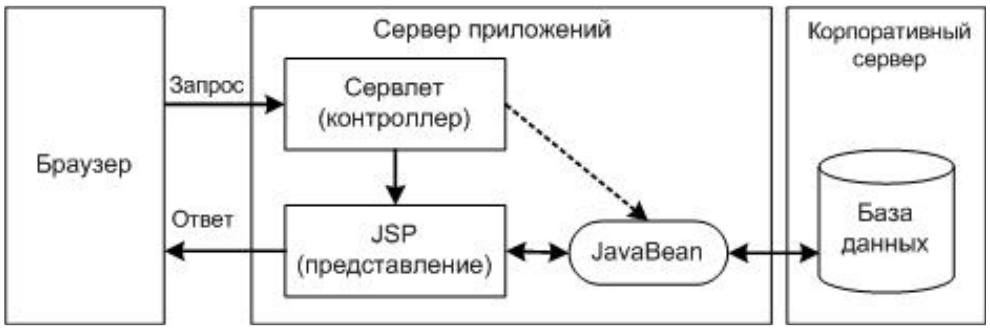


Рисунок 4.9 – Порядок работы *JSP Model 2*

Необходимо отметить, что в рамках этой модели страница *JSP* сама не реализует никакую логику, это входит в функции сервлета-контроллера. Страница *JSP* отвечает только за получение информации от компонента *JavaBean*, который был предварительно создан сервлетом, и за визуальное представление этой информации в удобном для клиента виде.

Архитектуры *JSP Model 2* в большей степени, чем архитектура *JSP Model 1*, соответствует идее отделения представления от содержания. Эта модель позволяет четко выделить отдельные части приложения и связанные с ними роли и обязанности персонала, занятого в разработке.

Чем сложнее разрабатываемая система, тем заметнее становятся преимущества архитектуры *JSP Model 2*.

Функционирование *JSP*

Работа со страницей *JSP* становится возможной только после ее преобразования в сервлет. В процессе трансляции как статическая, так и динамическая части *JSP* преобразуются в *Java*-код сервлета, который передает преобразованное содержимое браузеру через выходной поток веб-сервера.

Технология *JSP* является технологией серверной стороны, поэтому все процессы обработки *JSP* протекают на стороне сервера. Страница *JSP* – текстовый документ, который в соответствии со спецификацией *JSP* проходит две фазы:

1) фазу трансляции;

2) фазу выполнения.

При трансляции, которая выполняется один раз для каждой страницы *JSP*, создается или локализуется класс типа *Servlet*, реализующий *JSP*. Трансляция

JSP может производиться как отдельно, до ее использования, так и в процессе размещения *JSP* на веб-сервере или сервере приложений. Во второй фазе осуществляется обработка запросов и подготовка ответов.

Синтаксис *JSP*-страницы

Страницы *JSP* имеют комбинированный синтаксис: объединение стандартного синтаксиса, соответствующего спецификации *HTML*, и синтаксиса *JSP*, определенного спецификацией *Java Server Pages*. Синтаксис *JSP* определяет правила записи страниц *JSP*, состоящих из стандартных тегов *HTML* и тегов *JSP*.

Страницы *JSP*, кроме *HTML*-тегов, содержат теги *JSP* следующих категорий:

1) директивы (*directives*);
2) страницы (*pages*);
3) библиотеки тегов (*taglib*);
4) включения (*include*);
5) объявления (*declarations*);
6) скриптлеты (*scriptlets*);
7) выражения (*expressions*);
8) комментарии (*comments*).

Директивы *JSP*

Директивы обеспечивают глобальную информацию, касаются конкретных запросов, направляемых в *JSP*, и предоставляют информацию, необходимую на стадии трансляции.

Директивы всегда помещаются в начале *JSP*-страницы до всех остальных тегов, чтобы *parser* (анализатор) *JSP* при разборе текста в самом начале выделил глобальные инструкции. Таким образом, *JSP Engine* (среда исполнения *JSP*), анализируя код, создает из *JSP* сервлет. Директивы представляют собой сообщения контейнеру *JSP*.

Синтаксис директив *JSP* выглядит следующим образом:

<%@ директива имяАтрибута="значение" %>

Синтаксис задания директив на *XML*:

<*jsp:directive*.директива имяАтрибута="значение" />

Директива может иметь несколько атрибутов. В этом случае директива может быть повторена для каждого из атрибутов. В то же время пары «имяАтрибута=значение» могут располагаться под одной директивой с пробелом в качестве разделителя.

Существует три типа директив:

1) *page* (страница);

2) *taglib* (библиотека тегов);

3) *include* (включение).

Директива *page*

Директива *page* определяет свойства страницы *JSP*, которые воздействуют на транслятор. Порядок следования атрибутов в директиве *page* не имеет значения. Нарушение синтаксиса или наличие нераспознанных атрибутов приводит к ошибке трансляции. Примером директивы *page* может служить следующий код:

```
<%@ page buffer="none" isThreadSafe="yes" errorPage="/error.jsp" %>
```

Эта директива объявляет, что данная страница *JSP* не использует буферизацию, что возможно одновременное обращение к данной странице *JSP* многих пользователей и что поддерживается страница ошибок с именем *error.jsp*. Директива *page* может содержать информацию о странице:

```
<%@ page info = "JSP Sample 1" %>
```

Директива *taglib*

Директива *taglib* объявляет, что данная страница *JSP* использует библиотеку тегов, уникальным образом идентифицируя ее с помощью *URI*, и ставит в соответствие префикс тега, с помощью которого возможны действия в библиотеке.

Директива *taglib* имеет следующий синтаксис:

```
<%@ taglib uri="URI включаемой библиотеки тегов" prefix="имяПрефикса" %>
```

Префикс "*имяПрефикса*" используется при обращении к библиотеке. Пример использования библиотеки тегов *mytags*:

```
<%@ taglib uri="http://www.taglib/mytags" prefix="customs" %>
. . .
<customs:myTag>
```

В данном примере библиотека тегов имеет *URI*-адрес: *http://www.taglib/mytags*, в качестве префикса назначена строка *customs*, которая используется в странице *JSP* при обращении к элементам библиотеки тегов.

Директива *include*

Директива *include* позволяет вставлять текст или код в процессе трансляции страницы *JSP* в сервлет. Синтаксис директивы *include* имеет следующий вид:

```
<%@ include file="Относительный URI включаемой страницы" %>
```

Директива *include* имеет один атрибут – *file*. Она включает текст специфицированного ресурса в файл *JSP*. Эту директиву можно использовать для размещения стандартного заголовка об авторских правах на каждой странице *JSP*:

```
<%@ include file="copyright.html" %>
```

Контейнер *JSP* получает доступ к включаемому файлу. Если включаемый файл изменился, контейнер может перекомпилировать страницу *JSP*. Директива *include* рассматривает ресурс, например, страницу *JSP*, как статический объект.

Заданный *URI* обычно интерпретируется относительно *JSP*-страницы, на которой расположена ссылка, но, как и при использовании любых других относительных *URI*, можно задать системе положение интересующего ресурса относительно домашнего каталога веб-сервера добавлением в начало *URI* символа "/". Содержимое подключаемого файла обрабатывается как обычный текст *JSP* и поэтому может включать такие элементы, как статический *HTML*, элементы скриптов, директивы и действия.

Многие сайты используют небольшую панель навигации на каждой странице. В связи с проблемами использования фреймов *HTML* часто эта задача решается размещением небольшой таблицы сверху или в левой половине страницы, *HTML*-код которой многократно повторяется для каждой страницы сайта. Директива *include* – это наиболее естественный способ решения данной задачи, избавляющий разработчика от рутины копирования *HTML* в каждый отдельный файл.

Поскольку директива *include* подключает файлы в ходе трансляции страницы, то после внесения изменений в панель навигации требуется повторная трансляция всех использующих ее *JSP*-страниц. Если же подключенные файлы меняются довольно часто, можно использовать действие *jsp:include*, которое подключает файл в процессе обращения к *JSP*-странице.

Выражения *JSP*

Выражение в странице *JSP* – это исполняемое выражение, написанное на языке скрипта, указанного в объявлении *language* (как правило, *Java*). Результат

выражения *JSP*, имеющий обязательный тип *String*, направляется в стандартный поток вывода *out* с помощью текущего объекта *JspWriter*. Если результат выражения не может быть приведен к типу *String*, возникает либо ошибка трансляции, если проблема была выявлена на этапе трансляции, либо инициируется исключение *ClassCastException*, если несоответствие было выявлено в процессе выполнения запроса. Выражение имеет следующий синтаксис:

```
<%= текст выражения %>
```

Альтернативный синтаксис для выражений *JSP* при использовании *XML*:

```
<jsp:expression> текст выражения </jsp:expression>
```

Порядок выполнения выражений в странице *JSP* – слева направо. Если выражение появляется более чем в одном атрибуте времени выполнения, то оно выполняется слева направо в данном теге. Выражение должно быть полным выражением на определенном скрипте (как правило, *Java*).

Выражения выполняются во время работы протокола *HTTP*. Значение выражения преобразуется в строку и включается в соответствующую позицию файла *JSP*.

Выражения обычно используются для того, чтобы вычислить и вывести на экран строковое представление переменных и методов, определенных в блоке объявлений страницы *JSP* или полученных от компонентов *JavaBeans*, которые доступны из *JSP*. Следующий код выражения служит для отображения даты и времени запроса данной страницы:

```
Текущее время: <%= new java.util.Date () %>
```

Для того чтобы упростить выражения, существует несколько заранее определенных переменных, которые можно использовать. Наиболее часто используемые переменные:

1) *request, HttpServletRequest*;

2) *response, HttpServletResponse*;

3) *session, HttpSession* – ассоциируется с запросом, если таковой имеется;

4) *out, PrintWriter* – буферизированный вариант типа *JspWriter* для отсылки данных клиенту.

Неявные объекты

Неявные объекты (*implicit objects*) – это объекты, автоматически доступные как часть стандарта *JSP* без их специального объявления или импорта. Эти объекты, список которых представлен в таблице 4.5, можно использовать в коде *JSP*.

Таблица 4.5 – Неявные объекты

Наименование объекта	Тип объекта	Назначение
request (запрос)	*javax.servlet.HttpServletRequest*	Запрос, требующий обслуживания. Область видимости – запрос. Основные методы: *getAttribute, getParameter, getParameterNames, getParameterValues.* Таким образом, запрос *request* обеспечивает обращение к параметрам запроса через метод *getParameter*, типу запроса и входящим *HTTP*-заголовкам
response (ответ)	*javax.servlet.HttpServletResponse*	Ответ на запрос. Область видимости – страница. Поскольку поток вывода (см. *out* далее) буферизован, можно изменять коды состояния *HTTP* и заголовки ответов, даже если это недопустимо в обычном сервлете, но лишь в том случае, если какие-то данные вывода уже были отправлены клиенту
out (вывод)	*javax.servlet.jsp.JspWriter*	Объект, который пишет в выходной поток. Область видимости – страница. Основные методы: *clear, clearBuffer, flush, getBufferSize, getRemaining.* Необходимо помнить, размер буфера можно изменять и даже отключить буферизацию, изменяя значение атрибута *buffer* директивы *page*. Также необходимо обратить внимание, что *out* используется практически исключительно скриплетами, поскольку выражения *JSP* автоматически помещаются в поток вывода, что избавляет от необходимости явного обращения к *out*
pageContext (содержание страницы)	*javax.servlet.jsp.pageContext*	Содержимое *JSP*-страницы. Область видимости – страница. *pageContext* поддерживает доступ к полезным объектам и методам, обеспечивающим явный доступ реализации *JSP* к специфическим объектам. Основные методы: *getSession, getPage, findAttribute, getAttribute, getAttributeScope, getAttributeNamesInScope, getException*

Наименование объекта	Тип объекта	Назначение
session (сеанс)	*javax.servlet.HttpSession*	Объект типа *Session*, создаваемый для клиента, приславшего запрос. Область видимости – страница. Основные методы *getId, getValue, getValueNames, putValue.* Сессии создаются автоматически, и переменная *session* существует, даже если нет ссылок на входящие сессии. Единственным исключением является ситуация, когда разработчик отключает использование сессий, используя атрибут *session* директивы *page*. В этом случае ссылки на переменную *session* приводят к возникновению ошибок при трансляции *JSP*-страницы в сервлет
application (приложение)	*javax.servlet.ServletContext*	Контекст сервлета, полученный из объекта конфигурации сервлета при вызове методов: *getServletConfig* или *getContext*. Область видимости – приложение. Основные методы: *getMimeType*, *getRealPath*
config (конфигурация)	*javax.servlet.ServletConfig*	Объект *ServletConfig* текущей страницы *JSP*. Область видимости – страница. Основные методы: *getInitParameter*, *getInitParameterNames*
page (страница)	*java.lang.Object*	Экземпляр класса реализации текущей страницы *JSP*, обрабатывающий запрос. Область видимости – страница. Объект доступен, но, как правило, используется редко. По сути, является синонимом для *this* и не нужен при работе с *Java*
exception (исключение)	*java.lang.Throwable*	Объект *Throwable*, выводимый в страницу ошибок *error*-page. Область видимости – страница. Основные методы: *printStackTrace, toString*, *getMessage*, *getLocalizedMessage*

Создайте папку *views* в *WEB-INF* и в ней *login.jsp* (рисунок 4.10).

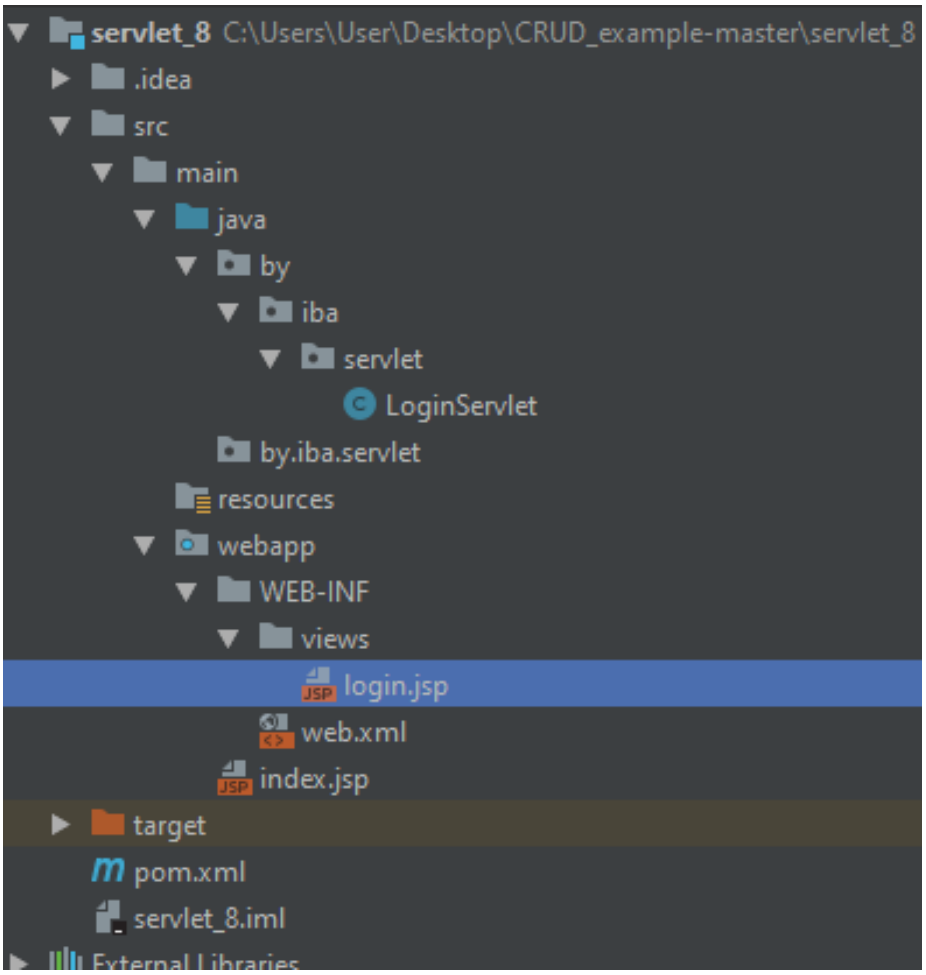


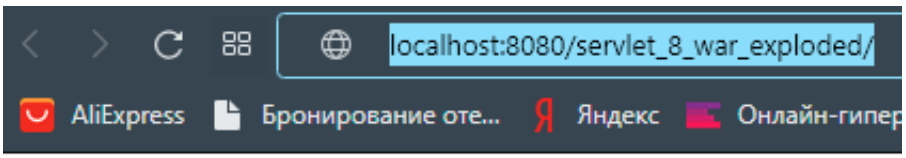
Рисунок 4.10 – Структура проекта

Создание *JSP*-страницы:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head><title>Title</title></head>
<body>
First JSP
<p>Today <%= new java.util.Date() %></p>
</body>
</html>
```

Изменим наш сервлет и получим (рисунок 4.11):

```
    @Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request,
response);
}
```



First JSP

Today Tue May 31 14:15:06 MSK 2022

Рисунок 4.11 – Страница с динамической информацией

Пересылка запроса (*forward*)

Когда запрос (*request*) браузера отправлен к *Servlet*, он может переслать запрос на другую страницу (или другой *Servlet*). Адрес в браузере пользователя также является ссылкой на первую страницу, но содержание страницы создается страницей пересылки. Пересылающая страница обязательно должна быть одной страницей (или *Servlet*) в вашем веб-приложении.

С пересылкой запроса (*forward*) вы можете использовать *request.setAttribute()* для передачи данных от страницы 1 к странице 2 (рисунок 4.12).

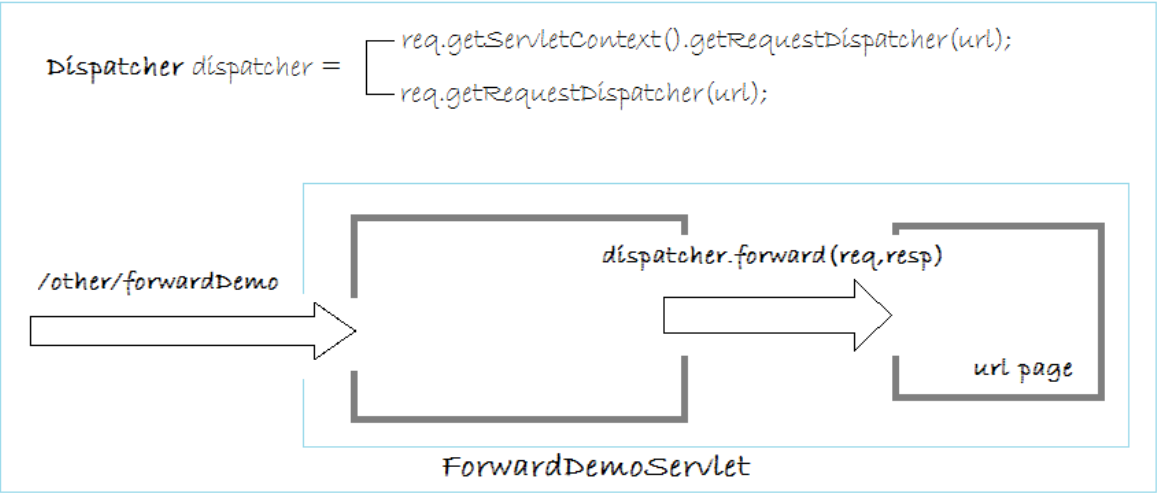


Рисунок 4.12 – Применение *forward*

Перенаправление (*redirect*) позволяет вам перенаправлять запросы на другие страницы, включая те, что за пределами веб-сайта.

Перенаправление запроса (*redirect*)

Когда запрос (*request*) отправлен от пользователя к *Servlet* (страница A), этот *Servlet* может направить запрос на другую страницу (страница B) и завершить свою задачу. Страница может быть перенаправлена на другую страницу в вашем приложении. Адрес в браузере пользователя теперь отображает ссылку на

страницу B. В отличие от пересылки (*forward*) с перенаправлением запроса (*redirect*) вы не можете использовать *request.setAttribute()* для передачи данных от страницы A на страницу B (рисунок 4.13).

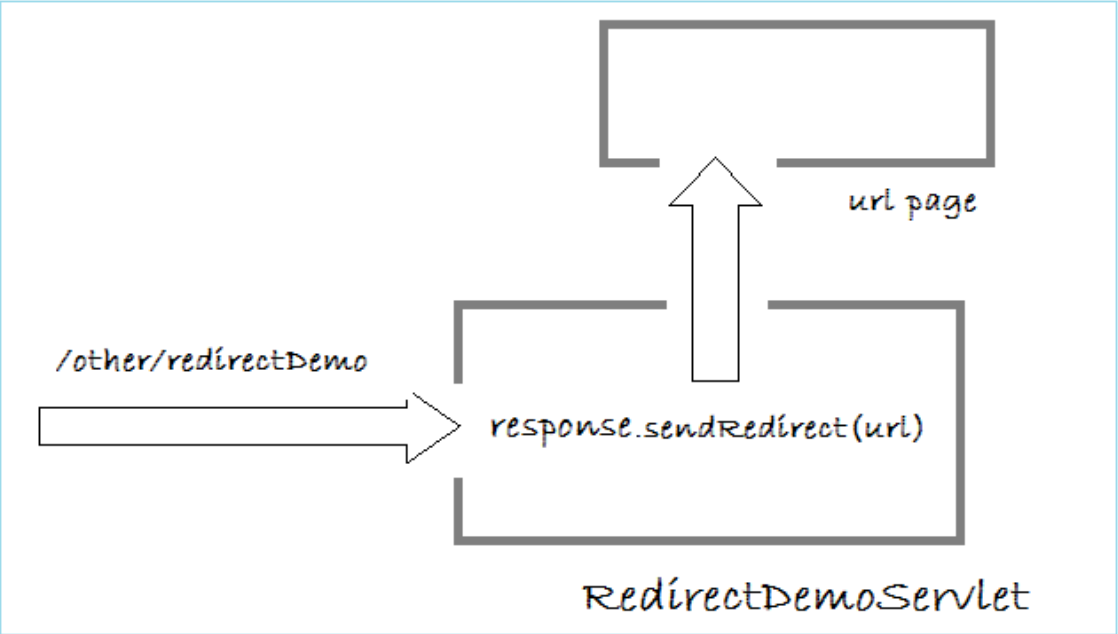


Рисунок 4.13 – Применение *SendRedirect*

Метод *forward()* целесообразно применять в следующих случаях:

1) когда мы используем метод *forward()*, запрос передается другому ресурсу на том же сервере для дальнейшей обработки;

2) в случае пересылки веб-контейнер обрабатывает весь процесс внутри, а клиент или браузер не задействованы;

3) когда *forward()* вызывается на объект *requestdispatcher*, мы передаем объекты запроса и ответа, чтобы наш старый объект запроса присутствовал на новом ресурсе, который будет обрабатывать наш запрос;

4) визуально мы не можем видеть перенаправленный адрес, он прозрачен.

5) использование метода *forward()* выполняется быстрее, чем перенаправление отправки;

6) когда мы перенаправляем с помощью *forward()* и хотим использовать одни и те же данные в новом ресурсе, мы можем использовать *request.setAttribute()*, поскольку у нас есть объект запроса.

Метод *sendRedirect()* целесообразно применять в следующих случаях:

1) в случае *sendRedirect()* запрос передается другому ресурсу на другой домен или на другой сервер для дальнейшей обработки;

2) при использовании *sendRedirect()* контейнер передает запрос клиенту или браузеру, поэтому *URL*-адрес, указанный внутри метода *sendRedirect()*, отображается как новый запрос клиенту;

3) в случае вызова *sendRedirect()* старые объекты запроса и ответа теряются, поскольку он рассматривается браузером как новый запрос;

4) в адресной строке мы можем видеть новый перенаправленный адрес. Его непрозрачность;

5) *sendRedirect()* медленнее, потому что требуется один дополнительный переход по представленной ссылке;

6) но в *sendRedirect()*, если мы хотим его использовать, мы должны хранить данные в сеансе или передавать вместе с *URL*.

Если вы хотите, чтобы управление передавалось на новый сервер или контекст, и оно рассматривалось как совершенно новая задача, тогда мы отправляемся на *sendRedirect()*. Как правило, переадресация должна использоваться, если операция может быть безопасно повторена при перезагрузке веб-страницы браузером, и это не повлияет на результат.

4.4 Передача параметров *HTTP*-запроса

В этом подразделе научимся получать параметры из *HTTP*-запроса. Допустим, мы хотели бы показать значение на *JSP*. Установим их как атрибут запроса. Атрибуты запроса могут быть доступны из представления.

В *LoginServlet.java* пишем следующее:

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.setAttribute("name", "I.O. Dima");
    request.setAttribute("password", "1111");
    request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request,
response);
}
```

Допишите в *WEB-INF\views\login.jsp*

My name is <%= request.getAttribute("name") %> and password is <%= request.getAttribute("password") %>

Запускаем и проверяем результат (рисунок 4.14).

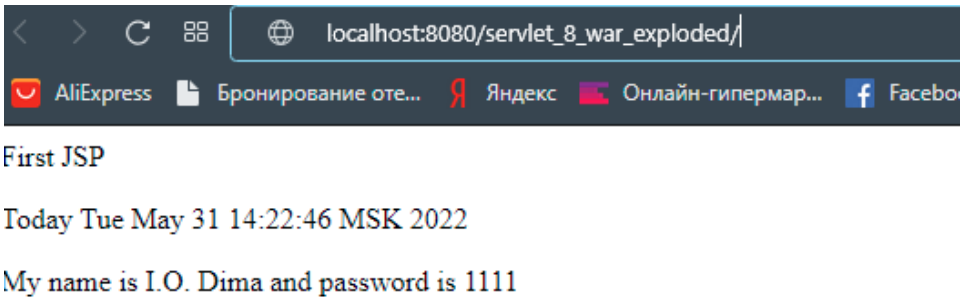


Рисунок 4.14 – Проверка работы с передачей параметров запроса

Страницы *JSP* могут получать отправленные данные, например, через параметры или в виде отправленных форм, так же как это происходит в сервлете. Для этого внутри страницы *JSP* доступен объект *request*, который позволяет получить данные посредством следующих методов:

– *getParameter(String param)*: возвращает значение определенного параметра, название которого передается в метод. Если указанного параметра в запросе нет, то возвращается значение *null*;

– *getParameterValues(String param)*: возвращает массив значений, который представляет определенный параметр. Если указанного параметра в запросе нет, то возвращается значение *null*.

```
<!DOCTYPE html> <html> <head> <meta charset="UTF-8" />
<title>User Info</title>
</head>
<body>
<p>Name: <%= request.getParameter("username") %></p>
<p>Country: <%= request.getParameter("country") %></p>
<p>Gender: <%= request.getParameter("gender") %></p>
 <h4>Selected courses</h4>
 <ul> <% String[] courses = request.getParameterValues("courses");
for(String course: courses){ out.println("<li>" + course + "</li>"); } %>
</ul>
</body>
</html>
```

Передача данных из сервлета в *JSP*

Нередко страница *JSP* обрабатывает запрос вместе с сервлетом. В этом случае сервлет определяет логику, а *JSP* – визуальную часть. И при обработке запроса сервлет может перенаправить дальнейшую обработку странице *JSP*. Соответственно может возникнуть вопрос, как передать данные из сервлета в *JSP*?

Есть несколько способов передачи данных из сервлета в *JSP*, которые заключаются в использовании определенного контекста или *scope*. Есть несколько контекстов для передачи данных:

– *request* (контекст запроса): данные сохраняются в *HttpServletRequest*;

– *session* (контекст сессии): данные сохраняются в *HttpSession*;

– *application* (контекст приложения): данные сохраняются в *ServletContext*.

Данные из контекста запроса доступны только в пределах текущего запроса. Данные из контекста сессии доступны только в пределах текущего сеанса, а данные из контекста приложения доступны постоянно, пока работает приложение.

Но вне зависимости от выбранного способа передача данных осуществляется с помощью метода *setAttribute(name, value)*, где *name* – строковое название данных, а *value* – сами данные, которые могут представлять различные данные.

Наиболее распространенный способ передачи данных из сервлета в *JSP* представляют атрибуты запроса, т. е. вызывается метод *setAttribute()* у объекта *HttpServletRequest*, который передается в сервлет. Этот метод устанавливает атрибут, который можно получить в *JSP*.

Для вывода атрибутов применяется специальный синтаксис *Expression Language* (*EL)*. В фигурные скобки {} заключается выводимое значение:

```
<p>Name: ${name}</p> <p>Age: ${age}</p>
```

Expression Language

Expression Language (или сокращенно *EL*) представляет собой компактный синтаксис для обращения к массивам, коллекциям, объектам и их свойствам внутри страницы *JSP*. Он довольн прост. Вставку открывает знак «$», затем в фигурные скобки {} заключается выводимое значение.

Откуда эти данные берутся? *EL* пытается найти значения для этих данных во всех доступных контекстах.

И *EL* просматривает все эти контексты в следующем порядке:

1) контекст страницы (данные сохраняются в *PageContext*);

2) контекст запроса;

3) контекст сессии;

4) контекст приложения.

Соответственно, если контексты запроса и сессии содержат атрибут с одним и тем же именем, то будет использоваться атрибут из контекста запроса.

Затем найденное значение конвертируется в строку и выводится.

```
<% pageContext.setAttribute("name", "Tom"); %>
<!DOCTYPE html>
<html>
<head> <meta charset="UTF-8">
<title>JSP Application</title>
</head>
<body>
<p>Name: ${name}</p>
</body>
</html>
```

Однако может сложиться ситуация, что сразу в нескольких контекстах одновременно содержатся данные с одним и тем же именем, например, "*name*". Соответственно *EL* будет получать данные в порядке просмотра контекстов, но, возможно, нам захочется выводить данные из какого-то определенного контекста. В этом случае перед названием данных мы можем указать название контекста: *pageScope*, *requestScope*, *sessionScope* или *applicationScope*. Например:

```
<% pageContext.setAttribute("name", "Bob"); %>
<!DOCTYPE html>
<html>
<head> <meta charset="UTF-8">
<title>JSP Application</title>
</head>
<body>
<p>Name: ${requestScope.name}</p>
</body>
</html>
```

Встроенные объекты *Expression Language*

По умолчанию *Expression Language* предоставляет ряд встроенных объектов, которые позволяют использовать различные аспекты запроса:

– *param*: объект, который хранит все переданные странице параметры;

– *paramValues*: хранит массив значений для определенного параметра;

– *header*: хранит все заголовки запроса;

– *headerValues*: предоставляет массив значений для определенного заголовка запроса;

– *cookie*: предоставляет доступ к временным пользовательским данным в отправленном запросе;

– *initParam*: возвращает значение для определенного параметра из элемента *context-param* из файла *web.xml*;

– *pageContext*: предоставляет доступ к объекту *PageContext*, который представляет контекст текущей страницы *JSP*.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>User Info</title>
</head>
<body>
<p>Name: ${param.name}</p>
<p>Age: ${param.age}</p>
</body> </html>
```

Через объект *param* здесь получаем из запроса значения параметров *name* и *age*. Значения для параметров можно передать как через строку запроса, так и через отправку формы.

4.5 Добавление форм фронт-составляющей

В файл *WEB-INF\views\login.jsp* добавим форму. Пользователь будет вводить данные. И по нажатии кнопки данные формы уходят в сервлет:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head><title>Title</title></head>
<body>
<form action="LoginServlet" method="POST">
    Name : <input name="name" type="text"/>
    <input type="submit"/></form>
</body>
</html>
```

Добавляем еще одну *JSP*. *WEB-INF\views\welcome.jsp*:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head><title> Welcome</title>
</head>
<body>
Welcome, ${name}
</body>
</html>
```

В *LoginServlet.java* измените код:

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request,
response);}
```

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
    request.setAttribute("name", request.getParameter("name"));
    request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request,
response);
}
```

Проверим работу динамической формы ввода (рисунки 4.15 и 4.16).

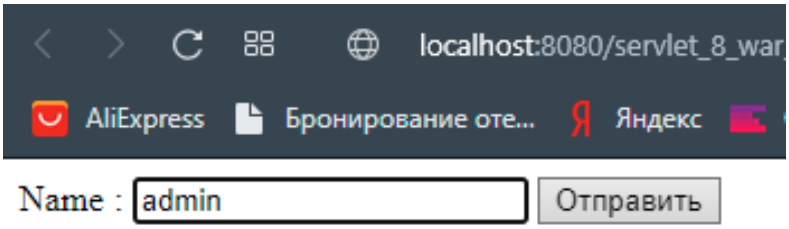


Рисунок 4.15 – Ввод значения формы запроса

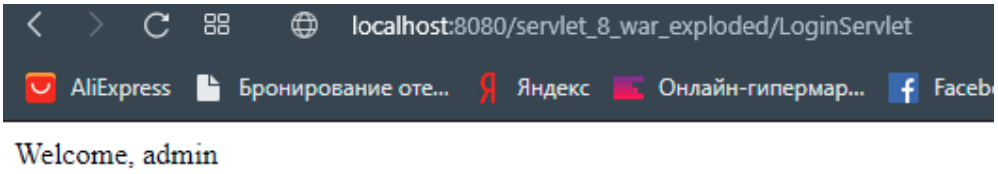


Рисунок 4.16 – Результат обработки запроса

4.6 Внедрение проверок

Добавим следующие функции. Пользователь вводит логин и пароль. Сервлет *LoginServlet.java* проверяет и перенаправляет на страницу приветствия, в случае ввода неверного значения выводит ошибку и выполняет переход на страницу ввода логина.

Добавим метод проверки в *LoginServlet.java*.

```
public boolean validateUser(String user, String password) {
    return user.equalsIgnoreCase("admin") && password.equals("admin");
}
```

Перепишем *doPost* в *LoginServlet.java*

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String name = request.getParameter("name");
    String password = request.getParameter("password");
```

```
    if (validateUser(name, password)) {
        request.getSession().setAttribute("name", name);
        request.getRequestDispatcher("/WEB-
INF/views/welcome.jsp").forward(request, response);
    } else {
        request.setAttribute("errorMessage", "Invalid Login and password!!");
        request.getRequestDispatcher("/WEB-
INF/views/login.jsp").forward(request, response);
    }
}
```

Изменим содержимое *login.jsp*. Добавим ввод пароля:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<p><font color="red">${errorMessage}</font>
</p>
<form action="LoginServlet" method="POST">
    <p>Name : <input name="name" type="text"/>
    </p>
    <p> Password : <input name="password" type="password"/>
    </p>
    <input type="submit"/></form>
</body>
</html>
```

Проверка отправки значений входа и регистрации пользователя (рисунки 4.17–4.19).

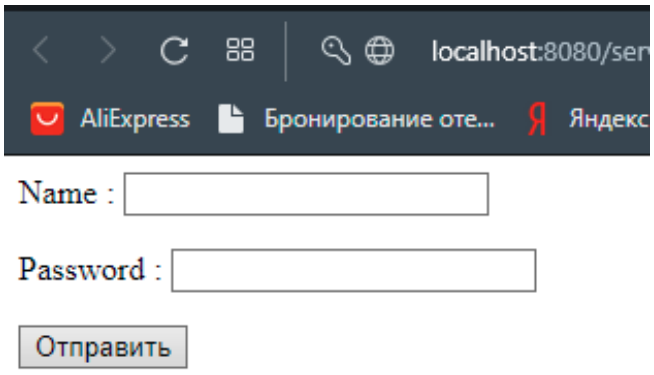


Рисунок 4.17 – Ввод значения формы запроса

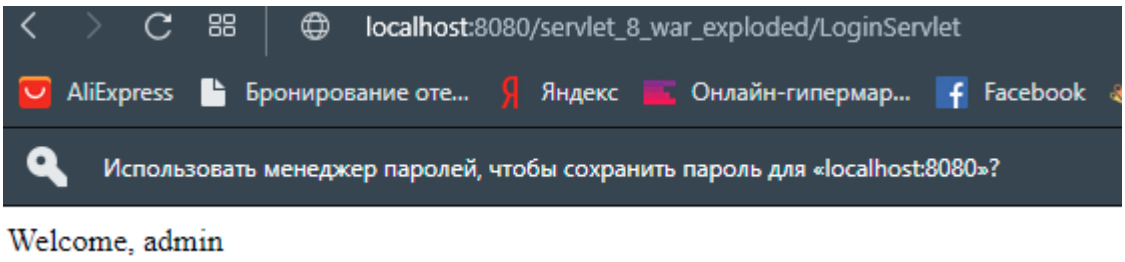


Рисунок 4.18 – Результат обработки запроса

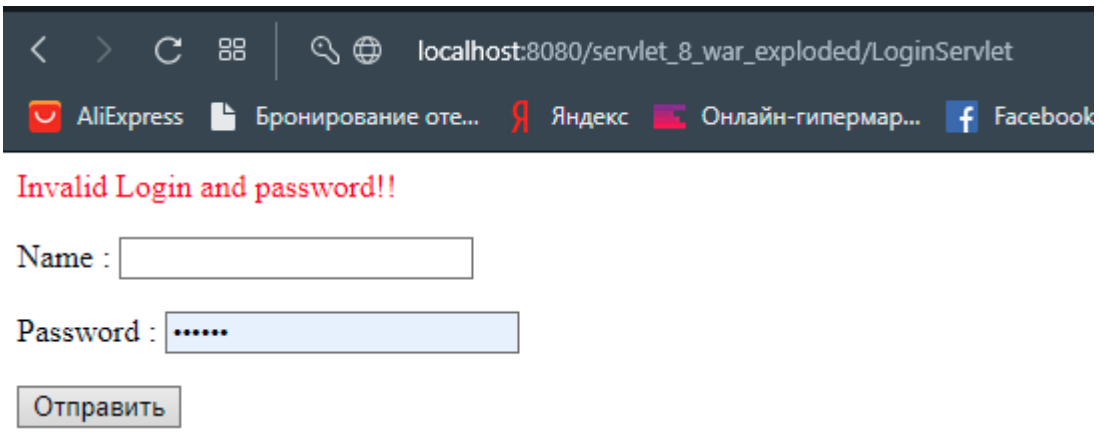


Рисунок 4.19 – Результат обработки запроса с ошибкой

Java Bean

Компоненты *JavaBeans* – это многократно используемые классы *Java*, позволяющие разработчикам существенно ускорять процесс разработкии веб-приложений путем их сборки из программных компонентов. *JavaBeans* и другие компонентные технологии привели к появлению нового типа программирования – сборки приложений из компонентов, при котором разработчик должен знать только сервисы компонентов; детали реализации компонентов не играют никакой роли.

Компоненты *JavaBeans* – это одноуровневые объекты, использующиеся для того, чтобы инкапсулировать в одном объекте сложный код, данные или и то, и другое. Компонент *JavaBean* может иметь свойства, методы и события, открытые для удаленного доступа.

Компонент *JavaBean* – это *Java*-класс, удовлетворяющий определенным соглашениям о наименовании методов и экспортируемых событий. Одним из важных понятий технологии *JavaBeans* является внешний интерфейс *properties* (свойства). *Property* – это пара методов (*getter* и *setter*), обеспечивающих доступ к информации о внутреннем состоянии компонента *JavaBean*.

Для обращения к компонентам *JavaBeans* на странице *JSP* необходимо использовать следующее описание тега в разделе *head*:

```
<jsp:useBean id="BeanID" [scope="page | request | session | application"]
class="BeanClass" />
```

BeanID определяет имя компонента *JavaBean*, являющееся уникальным в области видимости, заданной атрибутом *scope*. По умолчанию принимается область видимости *scope="page"*, т. е. текущая страница *JSP*.

Обязательный атрибут класса компонента *class* может быть описан следующим способом:

```
class="имя класса" [type="полное имя суперкласса"]
```

Свойство компонента *JavaBean* с именем *myBean* устанавливается тегом:

```
<jsp:setProperty name="myBean" property="Имя свойства" value="Строка или
выражение JSP" />
```

Для чтения свойства компонента *JavaBean* с именем *myBean* используется тег:

```
<jsp:getProperty name="myBean" property="Имя свойства"/>
```

В следующем листинге приведен пример компонента *JavaBean*, содержащего строку *mystr*, используемую в качестве свойств.

4.7 Работа с возвратом списка

Создадим список объектов типа *Person* и будем возвращать его на странице *welcome.jsp* для авторизованного пользователя.

Добавим пакет *model* и класс *Person* (рисунок 4.20).

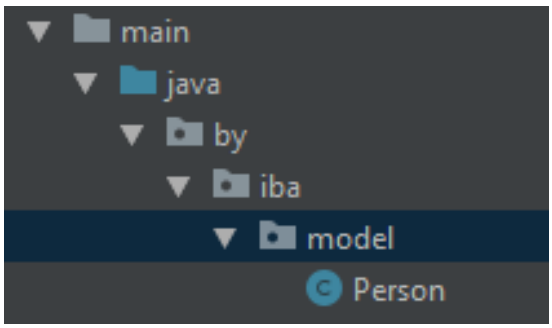


Рисунок 4.20 – Уровень объектной модели

```
public class Person {
    private String name;
    private String phone;
    private String email;
}
```

Сгенерируйте *set*, *get*, конструкторы, переопределите метод *toString*.

Затем создайте класс *ListService*:

```
package by.iba.model;
import java.util.ArrayList;
import java.util.List;
public class ListService {
    private static List<Person> groupList = new ArrayList();
    static {
        groupList.add(new Person("Anna", "+375291234567",
"anna.1.18@gmail.com"));
    }
    static public List<Person> retrieveList() {
        return groupList;
    }}
```

Добавим в *LoginServlet* следующий код:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    String name = request.getParameter("name");
    String password = request.getParameter("password");
    if (validateUser(name, password)) {
        request.getSession().setAttribute("name", name);
        request.setAttribute("group", ListService.retrieveList());
        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp")
                .forward(request, response);
    } else {
        request.setAttribute("errorMessage", "Invalid Login and
password!!");
        request.getRequestDispatcher("/WEB-INF/views/login.jsp")
                .forward(request, response);
    }}
```

Изменим *welcome.jsp*:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
<title> Welcome</title>
</head>
<body>
<H2>Welcome ${name}</H2>
</body>
</html>
```

В результате получим следующее (рисунок 4.21).

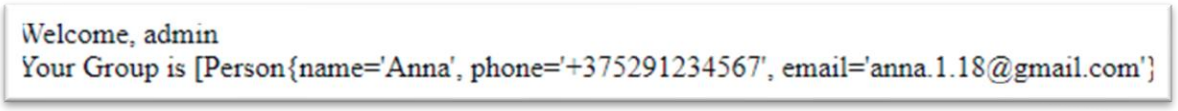


Рисунок 4.21 – Получение результата объектной модели

JSTL

JSP Standard Tag Library (*JSTL*) является стандартной библиотекой тегов, которая обеспечивает теги для управления поведением страницы, повторения команд управления, интернационализации тегов и тегов *SQL*.

JSTL является частью *JavaEE API* и большинства контейнеров *Servlet*. Но чтобы использовать *JSTL* в ваших страницах *JSP*, необходимо скачать библиотеки *JSTL* для вашего контейнера *Servlet*.

Обзор функций *JSTL*

На основании функций *JSTL* они делятся на пять категорий (таблица 4.6).

Таблица 4.6 – Функции *JSTL*

Функция	Описание
XML Tags	*XML*-теги используются для работы с *XML*-документами, например, для анализа *XML*, преобразования *XML*-данных и оценки выражений *XPath*
	<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>
JSTL Functions Tags	Теги *JSTL* предоставляют ряд функций, которые можно использовать для выполнения стандартных операций
	<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
Core Tags	Основные теги обеспечивают поддержку итерации, условной логики, исключения перехвата, *URL*, пересылки или перенаправления ответа и т. д.
	<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
Formattingand Localization Tags	Эти теги предназначены для форматирования чисел, дат и поддержки i18n через локали и пакеты ресурсов
	<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
SQL Tags	Теги *JSTL SQL* обеспечивают поддержку взаимодействия с реляционными базами данных, такими как *Oracle, MySql* и т. д. Используя теги *SQL*, вы можете выполнять запросы к базе данных
	<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>

Стандартные теги *JSTL* (*JSTL Core Tags*)

Таблица 4.7 – Теги *JSTL*

Тег	Описание
<c:out>	Чтобы написать что-либо на странице *JSP*, вы также можете использовать *EL* с этим тегом
<c:import>	То же самое, что и *<jsp:include>* или директива *include*
<c:redirect>	Перенаправить запрос на другой ресурс
<c:set>	Чтобы установить значение переменной в заданной области действия
<c:remove>	Чтобы удалить значение переменной в заданной области действия
<c:catch>	Чтобы перехватить исключение и поместить его в объект
<c:if>	Простая условная логика, используемая с *EL*, и вы можете использовать ее для обработки исключения из *<c:catch>*
<c:choose>	Простой условный тег, который устанавливает контекст для взаимоисключающих условных операций, отмеченных *<c:when>* и *<c:otherwise>*
<c:when>	Подтег *<c:choose>*, который включает его тело, если его условие оценивается как «истина»
<c:otherwise>	Подтег *<c:choose>*, который включает его тело, если его условие оценивается как «ложь»
<c:forEach>	Для итерации по коллекции
<c:forTokens>	Для итерации по токенам, разделенным разделителем
<c:param>	Используется с *<c:import>* для передачи параметров
<c:url>	Для создания *URL* с дополнительными параметрами строки запроса

4.8 Добавление *JSTL*

Предыдущий вывод можно сделать немного лучше, используя библиотеку тегов. Сначала необходимо подключить библиотеку. Она доступна по ссылке *https://search.maven.org/artifact/org.glassfish.web/jakarta.servlet.jsp.jstl.*

Скопируйте зависимость и разместите ее в файл *pom.xml* вашего проекта в раздел зависимостей и синхронизируйте файл.

Изменяем стартовую страницу:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<title> Title</title>
</head>
<body>
<H2>Welcome ${name}</H2>
<div>
    <table border="1">
        <caption>Список вашей группы</caption>
        <tr>
            <th>Имя</th>
            <th>Телефон</th>
            <th>email</th>
        </tr>
        <c:forEach items="${group}" var="person">
            <tr><td>${person.name}</td>
                <td>${person.phone}</td>
                <td>${person.email}</td>
            </tr>
        </c:forEach>
    </table>
</div>
</body>
</html>
```

Результат обработки формы (рисунок 4.22).

Welcome admin

Список вашей группы

Имя	Телефон	email
Anna	+375291234567	anna.1.18@gmail.com
Ivan	+375331114534	ivan.1.18@gmail.com
Nikolai	+3752998734534	nik.1.18@gmail.com

Рисунок 4.22 – Обработка формы

4.9 Реализация перенаправления между сервлетами

Плохо, что сервлет выполняет много работы: проверку прав пользователя и возврат списка. Разделим ответственность и создадим другой сервлет, который будет возвращать список группы. Взаимодействие будет следующим: если пользователь авторизовался в *LoginServlet*, *LoginServlet* перенаправляет запрос к новому сервлету *GroupListServlet*, который возвращает список группы.

Создадим сервлет *GroupListServlet*:

```
@WebServlet(name = "GroupListServlet", value = "/GroupListServlet")
public class GroupListServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;
    private ListService todoService = new ListService();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.setAttribute("group", ListService.retrieveList());
        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp")
                .forward(request, response);

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

        String nname = request.getParameter("nname");
        String nphone = request.getParameter("nphone");
        String nemail = request.getParameter("nemail");
        if (("".equals(nname)) || ("".equals(nphone)) || ("".equals(nemail))) {
            request.setAttribute("errorMessage", "Заполните все поля");
        } else {
            ListService.addPerson(new Person(nname, nphone, nemail));
        }
        request.setAttribute("group", ListService.retrieveList());
        request.getRequestDispatcher("/WEB-
INF/views/welcome.jsp").forward(request, response);

    }

}
```

Для взаимодействия сервлетов можно использовать два варианта.

Forward – работает внутри контейнера. Для браузера *URL* не меняется и кто реально отработал запрос, браузер не знает.

Redirect – браузеру посылается в ответе заголовок *Location*, в результате чего браузер делает новый запрос по переданному ему в этом заголовке *URL*, т. е. *URL* для браузера может меняться. Соответственно, теряется весь контекст, наработанный во время обработки до отправки *Redirect*.

Внесем изменения в *LoginServlet* в метод *Post*:

```
if (validateUser(name, password)) {
            request.getSession().setAttribute("name", name);
            response.sendRedirect(request.getContextPath() +
"/GroupListServlet");
// НЕТ ПАРАМЕТРОВ - всегда использует метод get
request.getRequestDispatcher("/GroupServlet")
//.forward(request, response);
```

4.10 Редактирование списка

Добавим новый метод *addPerson* для добавления объекта в список в классе *ListService*:

```
static public void addPerson(Person person) {
    groupList.add(new Person(person));
}
```

В класс *Person* добавим конструктор:

```
public Person(Person person) {
    name = person.name;
    phone = person.phone;
    email = person.email;
}
```

В класс *GroupListServlet* дописываем метод *doPost*:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String nname = request.getParameter("nname");
    String nphone = request.getParameter("nphone");
    String nemail = request.getParameter("nemail");
    if (("".equals(nname)) || ("".equals(nphone)) || ("".equals(nemail))) {
        request.setAttribute("errorMessage", "Заполните все поля");
    } else {
        ListService.addPerson(new Person(nname, nphone, nemail));
    }
    request.setAttribute("group", ListService.retrieveList());
    request.getRequestDispatcher("/WEB-INF/views/welcome.jsp").forward(request,
response);
}
```

В методе проверяем, что все поля заполнены. Если так, добавляем новый объект к списку.

Изменяем *welcome.jsp*. Теперь снизу надо добавить еще одну форму для ввода информации.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title> Title</title>
    <link href="webjars/bootstrap/4.3.1/css/bootstrap.min.css"
          rel="stylesheet">
    <style>
        .footer {
            position: absolute; bottom: 0;
            width: 100%;
            height: 60px;
            background-color: #f5f5f5;
        }
        .footer .container {
            width: auto;
            max-width: 680px;
            padding: 0 15px;
        }
    </style>
</head>
<body>
<nav role="navigation" class="navbar navbar-default">
    <div class="">
        <img src =
"https://www.kv.by/sites/default/files/user7743/logo_iba_group.jpg" width="50"
height="50">
    </div>
    <div class="navbar-collapse">
        <ul class="nav navbar-nav">
            <li class="active"><a href="#">Home</a></li>
        </ul>
        <ul class="nav navbar-nav navbar-right">
            <li><a href="LoginServlet">Login</a></li>
        </ul>
    </div>
</nav>
<div class="container">
    <H2>Welcome ${name}</H2>

    <table border="1">
        <caption>Список вашей группы</caption>
        <tr>
            <th>Имя</th>
            <th>Телефон</th>
            <th>email</th>
        </tr>
        <c:forEach items="${group}" var="person">
            <tr><td>  ${person.name}</td>
                <td>  ${person.phone}</td>
```

```
                <td>  ${person.email}</td>
            </tr>
        </c:forEach>
        </td>
        </td>
    </table>
    <p><font color="red">${errorMessage}</font></p>
    <form method="POST" action="GroupListServlet"> Новый :
        <p> Введите имя <input name="nname" type="text" /> </p>
        <p> Введите телефон <input name="nphone" type="text" /> </p>
        <p> Введите email <input name="nemail" type="text" /> </p>
        <input name="add" type="submit" />
    </form>
</div>
<footer class="footer">
    <div class="container">
        <p>2021 Все права защищены</p>
    </div>
</footer>
<script src="webjars/jquery/3.3.1/jquery.min.js"></script>
<script src="webjars/bootstrap/4.3.1/js/bootstrap.min.js"></script>
</body>
</html>
```

Получим следующий результат обработки запросов (рисунки 4.23 и 4.24).

Рисунок 4.23 – Заполнение полей списка

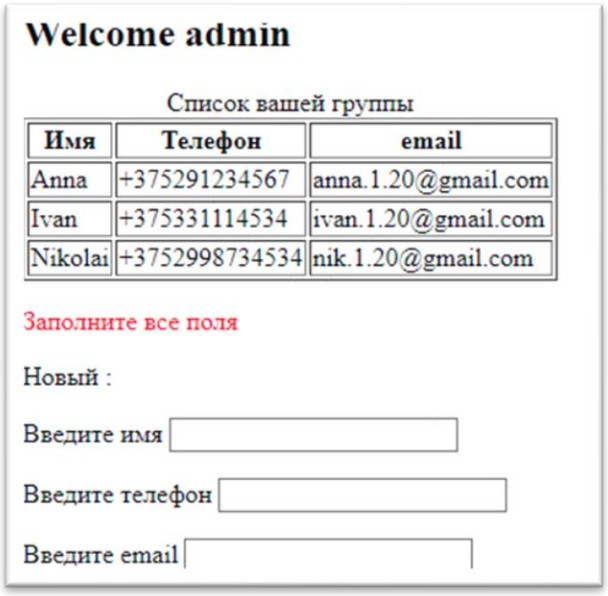


Рисунок 4.24 – Проверка заполнения полей формы

4.11 Добавление *JQuery* и *Bootstrap*

Если у вас нет *Bootstrap* и *JQuery*, скопируйте из репозитория *Maven* зависимости и добавьте в *pom.xml* (рисунки 4.25 и 4.26).

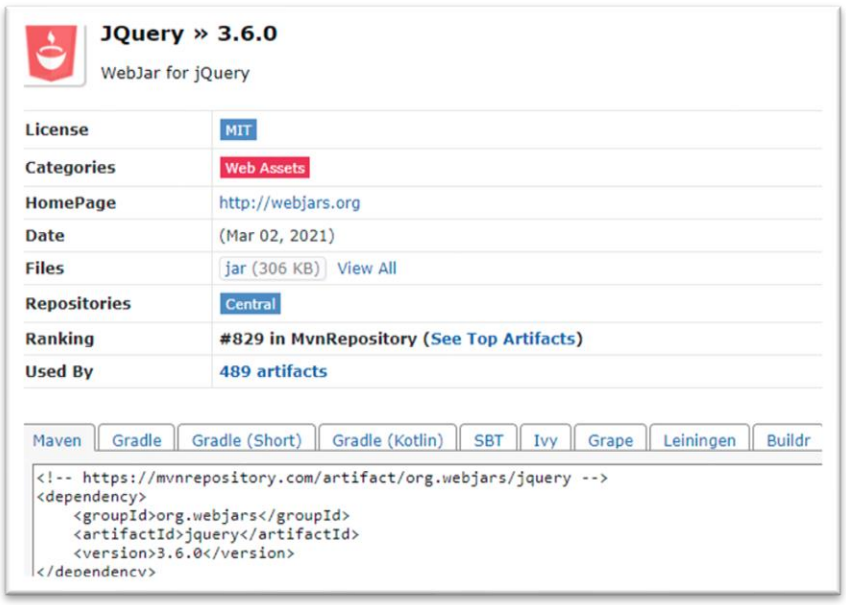


Рисунок 4.25 – Библиотека *JQuery*

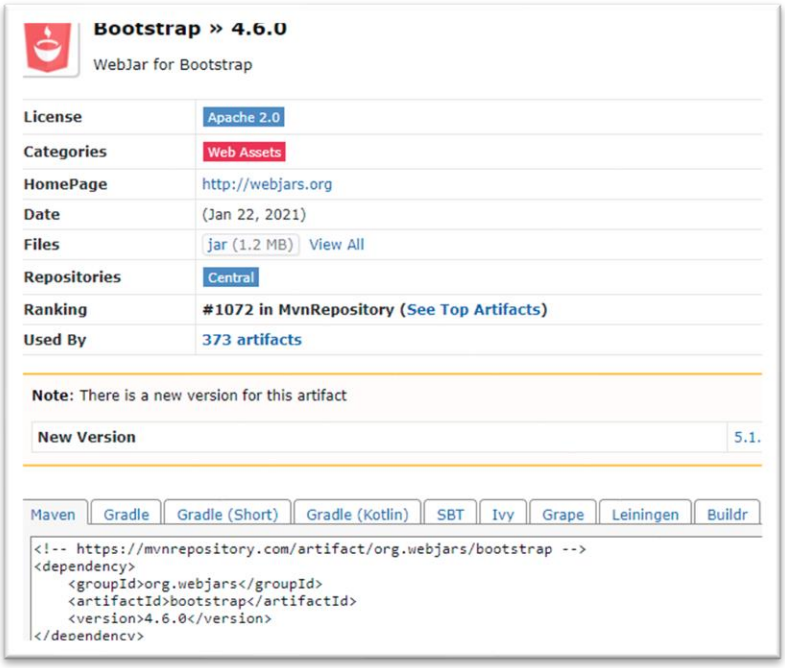


Рисунок 4.26 – Библиотека *Bootstrap*

Перепишем *welcome.jsp*. Вы можете сделать все на ваше усмотрение.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title> Title</title>
    <link href="webjars/bootstrap/4.3.1/css/bootstrap.min.css"
          rel="stylesheet">
    <style>
        .footer {
            position: absolute; bottom: 0;
            width: 100%;
            height: 60px;
            background-color: #f5f5f5;
        }
        .footer .container {
            width: auto;
            max-width: 680px;
            padding: 0 15px;
        }
    </style>
</head>
<body>
<nav role="navigation" class="navbar navbar-default">
    <div class="">
        <img src =
```

```
"https://www.kv.by/sites/default/files/user7743/logo_iba_group.jpg"
width="50" height="50">
    </div>
    <div class="navbar-collapse">
        <ul class="nav navbar-nav">
            <li class="active"><a href="#">Home</a></li>
        </ul>
        <ul class="nav navbar-nav navbar-right">
            <li><a href="LoginServlet">Login</a></li>
        </ul>
    </div>
</nav>
<div class="container">
    <H2>Welcome ${name}</H2>
    <table border="1">
        <caption>Список вашей группы</caption>
        <tr>
            <th>Имя</th>
            <th>Телефон</th>
            <th>email</th>
        </tr>

        <c:forEach items="${group}" var="person">
            <tr><td>${person.name}</td>
                <td>  ${person.phone}</td>
                <td>   ${person.email}</td>
            </tr>
        </c:forEach>
        </td>
        </td>
    </table>

    <p><font color="red">${errorMessage}</font></p>
    <form method="POST" action="GroupListServlet"> Новый :
        <p> Введите имя <input name="nname" type="text" /> </p>
        <p>    Введите телефон <input name="nphone" type="text" /> </p>
        <p> Введите email <input name="nemail" type="text" /> </p>
        <input name="add" type="submit" />
    </form>
</div>
<footer class="footer">
    <div class="container">
        <p>2021 Все права защищены</p>
    </div>
</footer>
<script src="webjars/jquery/3.3.1/jquery.min.js"></script>
<script src="webjars/bootstrap/4.3.1/js/bootstrap.min.js"></script>
</body>
</html>
```

Проверка переходов формы (рисунок 4.27).

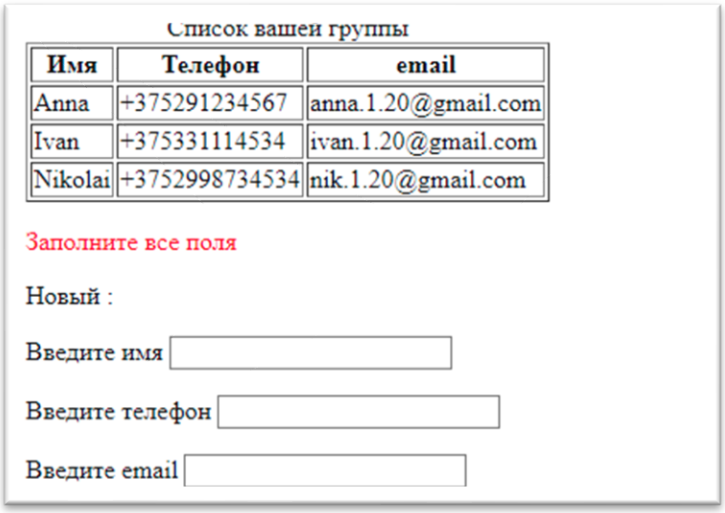


Рисунок 4.27 – Работа страницы с подключенными библиотеками

Назначение фильтров

Мы можем использовать фильтры для следующих задач:

– работа с ответами сервера перед тем, как они будут переданы клиенту;

– перехватывание запросов от клиента перед тем, как они будут отправлены на сервер.

Обычно, когда пользователь запрашивает веб-страницу, запрос будет отправлен на сервер (*Server*). Он должен проходить через фильтры (*Filter*) до того, как дойти до запрошенной страницы, как показано на рисунках 4.28 и 4.29.

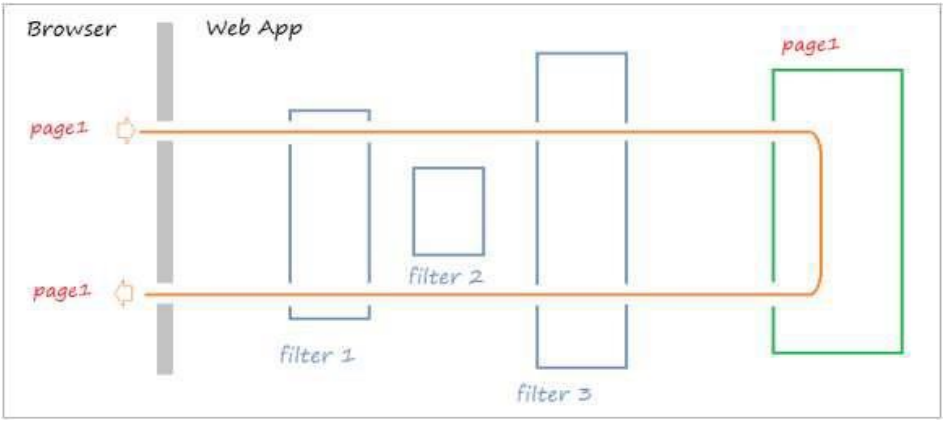


Рисунок 4.28 – Работа без применения фильтра

Однако существуют ситуации запроса пользователя, которые не проходят все уровни *Filter*.

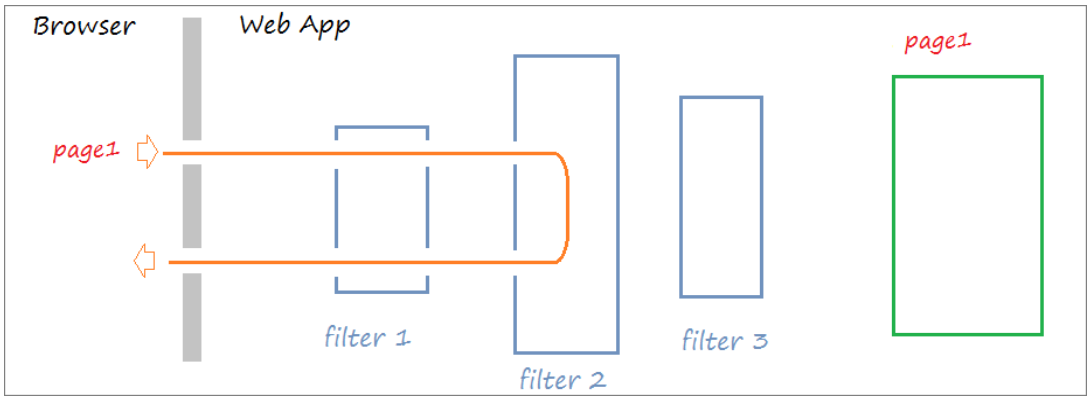


Рисунок 4.29 – Работа с применением фильтра

Ситуация, когда пользователь отправляет запрос одной страницы (*page 1*), этот запрос проходит через *Filter*.

Пример ситуации:

1) Пользователь посылает запрос, чтобы посмотреть страницу с личной информацией.

2) Запрос будет отправлен на *Server*.

3) Запрос проходит *Filter*, который записывает информацию в файл *log*.

4) Запрос к *Filter* проверяет, вошел ли пользователь в систему. *Filter* проверил и увидел, что пользователь не вошел в систему, тогда он перенаправляет запрос на страницу входа в систему пользователя (рисунок 4.30).

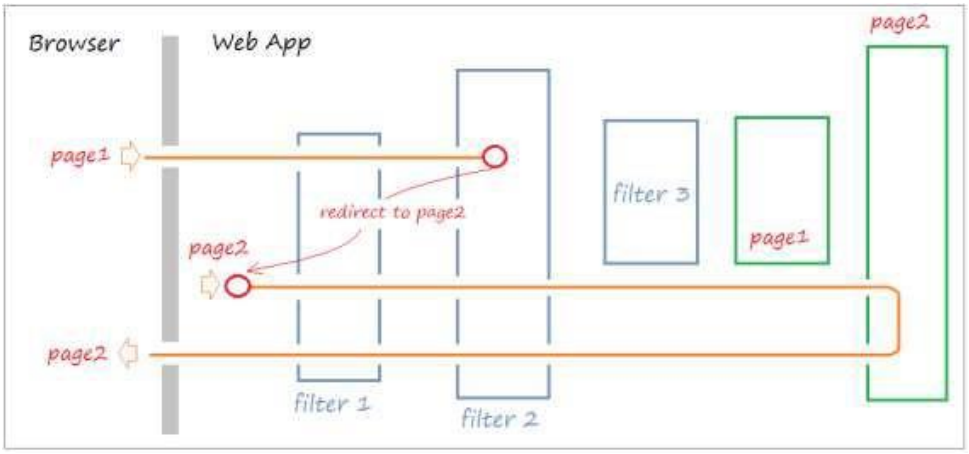


Рисунок 4.30 – Последовательность срабатывания фильтра

На самом деле *Filter* может быть использован для кодирования веб-сайта (*encoding*). Например, установить кодировку *UTF-8* для страницы. Открыть и закрыть соединение к *Database* и подготовить транзакцию *JDBC* (*JDBC Transaction*).

Сервлетные фильтры

Сервлетный фильтр, в соответствии со спецификацией, это *Java*-код, пригодный для повторного использования и позволяющий преобразовать содержание *HTTP*-запросов, *HTTP*-ответов и информацию, содержащуюся в заголовках *HTML*. Сервлетный фильтр занимается предварительной обработкой запроса, прежде чем тот попадает в сервлет, и/или последующей обработкой ответа, исходящего из сервлета.

Сервлетные фильтры могут:

– перехватывать инициацию сервлета прежде, чем сервлет будет инициирован;

– определить содержание запроса прежде, чем сервлет будет инициирован;

– модифицировать заголовки и данные запроса, в которые упаковывается поступающийзапрос;

– модифицировать заголовки и данные ответа, в которые упаковывается получаемый ответ;

– перехватывать инициацию сервлета после обращения к сервлету.

Сервлетный фильтр может быть конфигурирован так, что он будет работать с одним сервлетом или группой сервлетов. Основой для формирования фильтров служит интерфейс *javax.servlet.Filter*, который реализует три метода:

– *void init (FilterConfig config) throws ServletException*;

– *void destroy()*;

– *void doFilter* (*ServletRequest request, ServletResponse response, FilterChain chain*) *throwsIOException, ServletException*.

Метод *init* вызывается прежде, чем фильтр начинает работать, и настраивает конфигурационный объект фильтра. Метод *doFilter* выполняет непосредственно работу фильтра.Таким образом, сервер вызывает *init* один раз, чтобы запустить фильтр в работу, а затем вызывает *doFilter* столько раз, сколько запросов будет сделано непосредственно к данному фильтру. После того как фильтр заканчивает свою работу, вызывается метод *destroy*.

Пример простого фильтра:

```
package common;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FilterConnect implements Filter {
    private FilterConfig config = null;
    private boolean active = false;
public void init (FilterConfig config) throws ServletException
    {
        this.config = config;
        String act = config.getInitParameter("active");
        if (act != null)
            active = (act.toUpperCase().equals("TRUE"));
    }
public void doFilter (ServletRequest request, ServletResponse response,
    FilterChain chain)throws IOException,
    ServletException
    {
        if (active) {
// Здесь можно вставить код для обработки
        }
        chain.doFilter(request, response);
    }
public void destroy()
    {
        config = null;
    }
}
```

Интерфейс *FilterConfig* содержит метод для получения имени фильтра, его параметров инициации и контекста активного в данный момент сервлета. С помощью своего метода *doFilter* каждый фильтр получает текущий запрос *request* и ответ *response*, а также *FilterChain,* содержащий список фильтров, предназначенных для обработки.

В *doFilter* фильтр может выполнять с запросом и ответом следующие действия: собирать данные или упаковывать объекты для придания им нового поведения. Затем фильтр вызывает *chain.doFilter*, чтобы передать управление следующему фильтру. После возвращения этого вызова фильтр может по окончании работы своего метода *doFilter* выполнить дополнительную работу над полученным ответом. К примеру, сохранить регистрационную информацию об этом ответе.

После того как класс-фильтр откомпилирован, его необходимо установить в контейнер и «приписать» (*map*) к одному или нескольким сервлетам. Объявление и подключение фильтра отмечается в дескрипторе поставки *web.xml* внутри элементов <*filter*> и <*filter-mapping*>. Для подключения фильтра к сервлету необходимо использовать вложенные элементы <*filter-name*> и <*servlet-name*>.

Объявление фильтра:

```
<filter>
    <filter-name>FilterName</filter-name>
    <filter-class>FilterConnect</filter-class>
    <init-param>
        <param-name>active</param-name>
        <param-value>true</param-true>
    </init-param>
</filter>
```

Подключение фильтра к сервлету.

```
<filter-mapping>
    <filter-name>FilterName</filter-name>
    <servlet-name>ServletName</servlet-name>
</filter-mapping>
```

В представленном коде дескриптора поставки *web.xml* объявлен класс-фильтр *FilterConnect* с именем *FilterName*. Фильтр имеет параметр инициализации *active*, которому присвивается значение *true*. Фильтр *FilterName* в разделе <*filter-mapping*> подключен к сервлету *ServletName*.

Порядок, в котором контейнер строит цепочку фильтров для запроса, определяется следующими правилами:

– цепочка, определяемая *urlPattern*, выстраивается в том порядке, в котором встречаются соответствующие описания фильтров в *web.xml*;

– последовательность сервлетов, определенных с помощью <*servlet-name*>, также выполняется в той последовательности, в которой эти элементы встречаются в дескрипторе поставки *web.xml*.

Для связи фильтра со страницами *HTML* или группой сервлетов необходимо использовать тег <*url-pattern*>.

Для подключения фильтра к *HTML*-страницам используем следующий код:

```
<filter-mapping>
    <filter-name>FilterName</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
```

Использование дополнительных ресурсов

В отдельных случаях недостаточно вставить в сервлет фильтр или даже цепочку фильтров, а необходимо обратиться к другому сервлету, странице *JSP*, документу *HTML*, *XML* или другому ресурсу. Если требуемый ресурс находится в том же контексте, что и сервлет, который его вызывает, то для получения ресурса необходимо использовать метод

```
public RequestDispatcher getRequestDispatcher(String path);
```

представленный в интерфейсе *ServletRequest*. Здесь *path* – это путь к ресурсу относительно контекста. Например, необходимо обратиться к сервлету *Connect*:

```
RequestDispatcher rd = request.getRequestDispatcher("Connect");
```

Если ресурс находится в другом контексте, то необходимо предварительно получить контекст методом

```
public ServletContext getContext(String uripath);
```

интерфейса *ServletContext*, а потом использовать метод

```
public RequestDispatcher getRequestDispatcher (String uripath);
```

интерфейса *ServletContext*. Здесь путь *uripath* должен быть абсолютным, т. е. начинаться с наклонной черты /. Например,

```
    RequestDispatcher rd = config.getServletContext() .getContext("/prod").
getRequestDispatcher("/prod/Customer");
```

Если требуемый ресурс – сервлет, помещенный в контекст под своим именем, то для его получения можно обратиться к методу

```
RequestDispatcher getNamedDispatcher (String name);
```

Все три метода возвращают *null*, если ресурс недоступен или сервер не реализует интерфейс *RequestDispatcher*. Как видно из описания методов, к другим ресурсам можно обратиться только через объект типа *RequestDispatcher*, который предлагает два метода обращения к ресурсу. Первый метод

```
public void forward (ServletRequest request, ServletResponse response);
```

просто передает управление другому ресурсу, предоставив ему свои аргументы *request* и *response*. Вызывающий сервлет выполняет предварительную обработку объектов *request* и *response* и передает их вызванному сервлету или другому ресурсу, который окончательно формирует ответ *response* и отправляет его клиенту, или опять вызывает другой ресурс. Например:

```
if (rd != null)
rd.forward
(request, response); else
response.sendError (HttpServletResponse.SC_NO_CONTENT);
```

Вызывающий сервлет не должен выполнять какую-либо отправку клиенту до обращения к методу *forward()*, иначе будет выброшено исключение класса *IllegalStateException*. Если же вызывающий сервлет уже что-то отправлял клиенту, то следует обратиться ко второму методу:

```
public void include (ServletRequest request, ServletResponse response);
```

Этот метод вызывает ресурс, который на основании объекта *request* может изменить тело объекта *response*. Но вызванный ресурс не может изменить заголовки и код ответа объекта *response*. Это естественное ограничение, поскольку вызывающий сервлет мог уже отправить заголовки клиенту. Попытка вызванного ресурса изменить заголовок будет просто проигнорирована. Можно сказать, что метод *include* выполняет такую же работу, как вставки на стороне сервера *SSI (Server Side Include).*

4.12 Реализация фильтра

Сделайте следующее. Запустите приложение и на открывшейся странице вы получите данные списка группы, чего не должно быть без ввода пароля.

Фильтр гарантирует, что пользователь не может попасть на страницу списка группы без входа в систему. Сделаем это, добавляя фильтр для перехвата всех запросов, соответствующих шаблону.

Создайте пакет для фильтров *Filter* (рисунки 4.31 и 4.32).

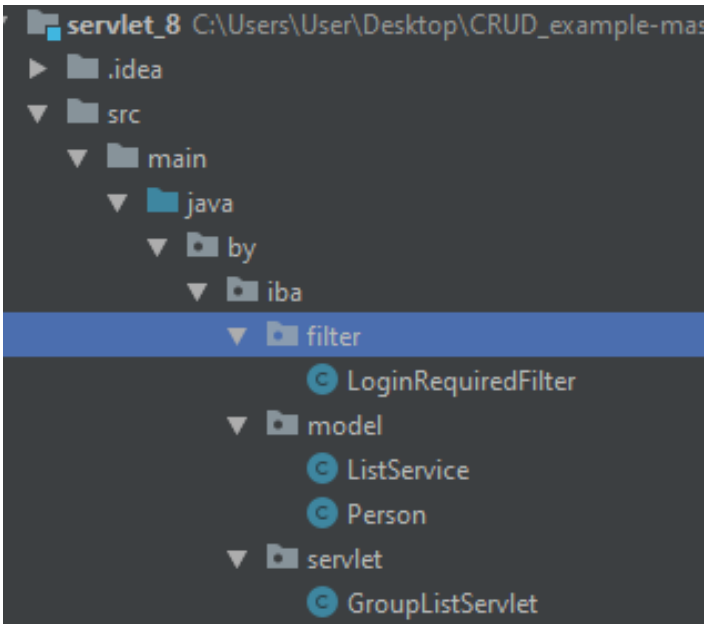


Рисунок 4.31 – Структура пакета фильтра

И создайте такой фильтр.

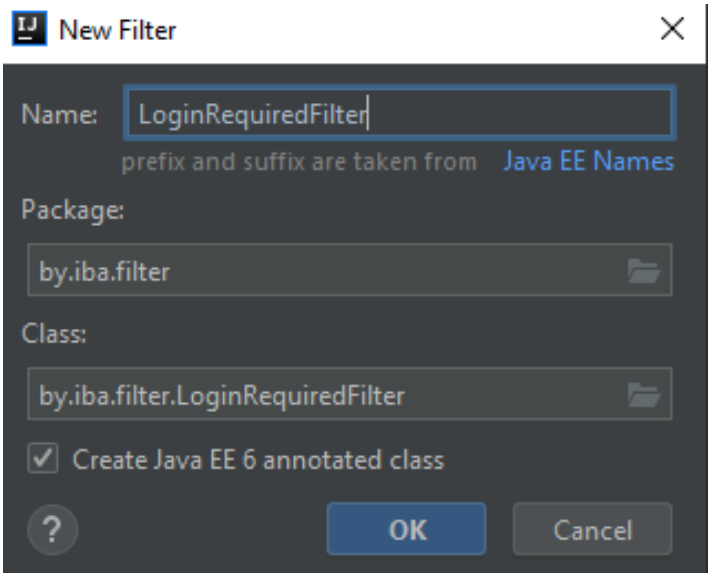


Рисунок 4.32 – Добавление фильтра

Должен появиться сгенерированный код:

```
@WebFilter(filterName = "LoginRequiredFilter")
public class LoginRequiredFilter implements Filter {
    public void destroy() {
    }
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
        chain.doFilter(req, resp);
    }
    public void init(FilterConfig config) throws ServletException {

    }
}
```

Фильтр должен сработать перед сервлетом *GroupListServlet*. Будем проверять имя пользователя. Если оно соответствует *admin*, пропускаем дальше (т. е. на сервлет *GroupListServlet*), если нет – переходим на *LoginServlet*.

Заменим код фильтра:

```
@WebFilter(filterName = "LoginRequiredFilter", urlPatterns =
"/GroupListServlet")
public class LoginRequiredFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        if ("admin".equals(request.getSession().getAttribute("name"))) {
            chain.doFilter(req, resp);
        } else {
            request.getSession().invalidate();
            request.getRequestDispatcher("LoginServlet").forward(req, resp);
        }
    }

    public void init(FilterConfig config) throws ServletException {
    }
}
```

Проверим.

Как видите, адресация у *GroupListServlet* не проходит без авторизации (рисунок 4.33).

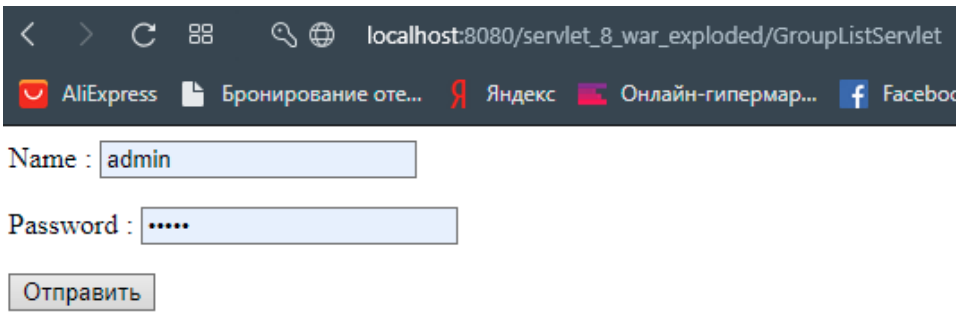


Рисунок 4.33 – Проверка фильтра на странице авторизации

Пользователь должен иметь возможность выйти. Поэтому допишем еще один серевлет *Logout* (рисунок 4.34).

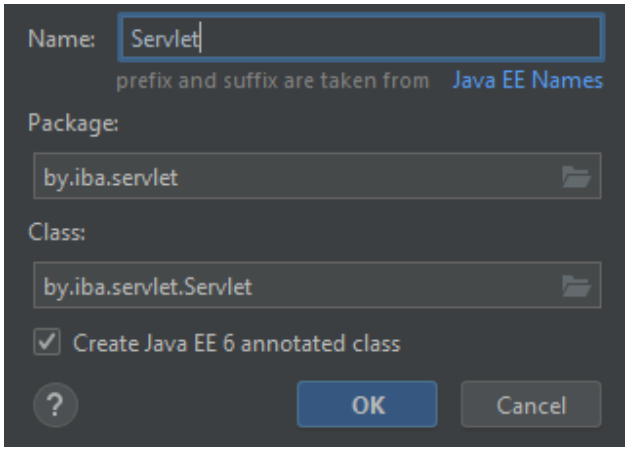


Рисунок 4.34 – Создание сервлета *Logout*

Добавим код:

```
@WebServlet(name = "LogoutServlet", value = "/LogoutServlet")
public class LogoutServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.getSession().invalidate();
        request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(
                request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    }
}
```

Добавим ссылку для выхода на *welcome.jsp*.

```
ul class="nav navbar-nav navbar-right">
      <li><a href="LogoutServlet">Logout</a></li></ul>
```

4.13 Конфигурация БД

Создадим БД, используя следующий скрипт:

```
USE infodb;
CREATE TABLE users (
id INTEGER AUTO_INCREMENT PRIMARY KEY,
login VARCHAR(20),
passw VARCHAR(12));
INSERT INTO users (login, passw) VALUES ('admin', 'admin');
```

Подключим БД с применением конфигурационого файла *Resource Bandle* (рисунки 4.35–4.37).

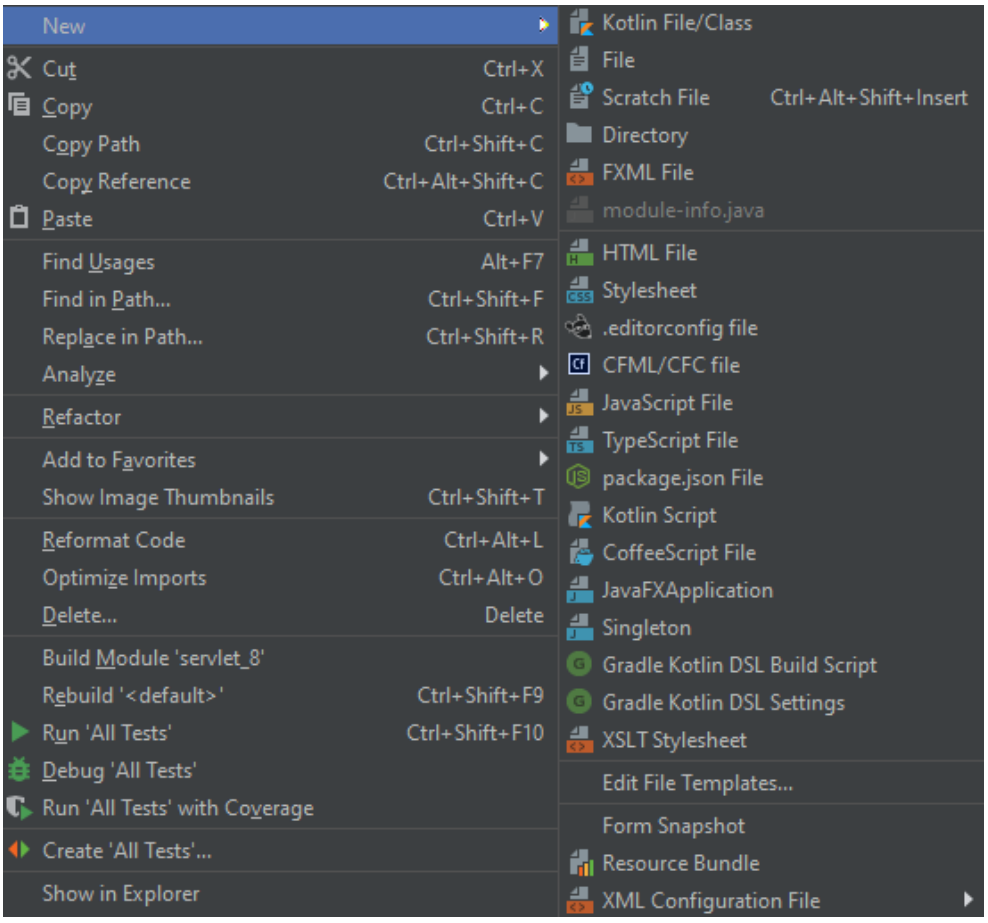


Рисунок 4.35 – Выбор *Resource Bandle*

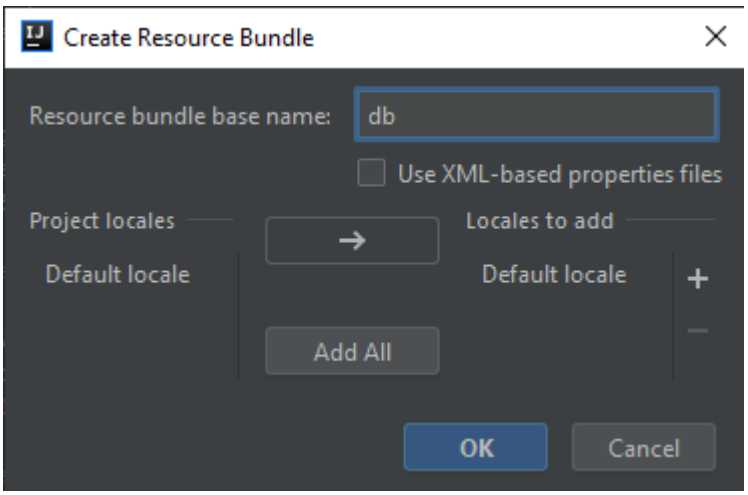


Рисунок 4.36 – Присвоение имени *Resource Bandle*

Рисунок 4.37 – Проверка создания *Resource Bandle*

Класс *ConnectorDB* использует файл ресурсов *db.properties*, в котором хранятся, как правило, параметры подключения к БД, такие, как логин и пароль доступа. Например:

```
db.driver   = com.mysql.jdbc.Driver
db.user = root
db.password = root
db.poolsize = 32
db.url = jdbc:mysql://localhost:3306/infodb
```

После этого создайте пакет *util* и в нем класс *ConnectorDB* (рисунок 4.38).

Рисунок 4.38 – Слой подключения к БД

```
    public static Connection getConnection() throws SQLException {
        ResourceBundle resource = ResourceBundle.getBundle("db",
                Locale.getDefault());
        String url = resource.getString("db.url");
        String user = resource.getString("db.user");
        String pass = resource.getString("db.password");

        return DriverManager.getConnection(url, user, pass);
    }
}
```

В таком случае получение соединения с БД сведется к вызову.

```
Connection cn = ConnectorDB.getConnection();
```

Когда мы будем проверять валидность пользователя, нам понадобится объект *User*. Создайте пакет *model* (рисунок 4.39):

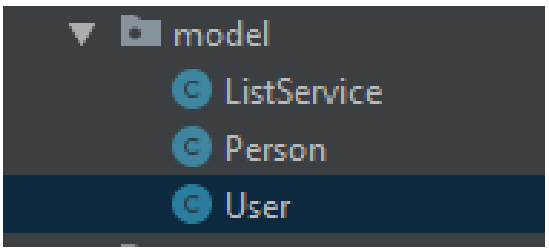


Рисунок 4.39 – Создание объектной модели

```
package by.iba.model;
public class User {

    private int id; private String login; private String passw;
    public User(int id, String login, String passw) {
        this.id = id; this.login = login; this.passw = passw;
    }
/Конструкторы getters@setters
    }
}
```

Осуществим оптимизацию кода посредством библиотеки *Lombok*, позволяющей сократить количество шаблонного кода, который нужно писать на *Java*.

Добавление в проект очень простое, достаточно добавить зависимости (рисунок 4.40).

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.18</version>
    <scope>provided</scope>
</dependency>
```

Затем подключить *plugin*. При использовании *Lombok* наш исходный код не будет валидным кодом *Java*. Поэтому потребуется установить плагин для *IDE*, иначе среда разработки не поймет, с чем имеет дело.

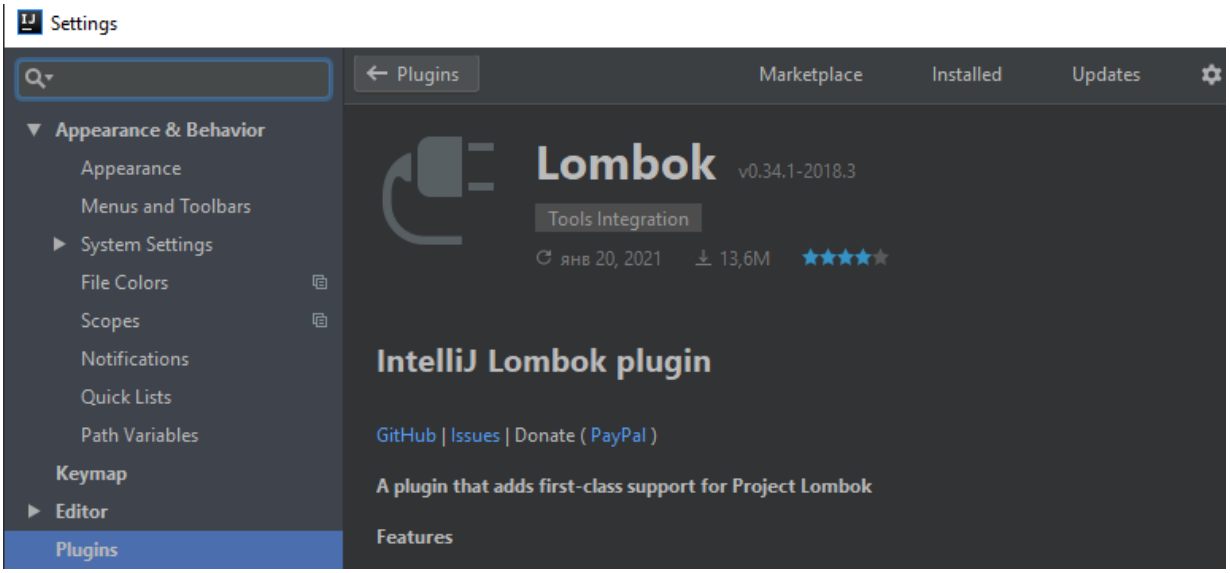


Рисунок 4.40 – Проверка плагина *Lombok*

Перечень аннотаций, используемых плагином *Lombok*:

@Data – просто удобная аннотация, которая применяет сразу несколько аннотаций *Lombok*.

@ToString генерирует реализацию для метода *toString()*, которая состоит из аккуратного представления объекта: имя класса, все поля и их значения.

@EqualsAndHashCode генерирует реализации *equals* и *hashCode*, которые по умолчанию используют нестатические и нестационарные поля, но настраиваются.

@Getter / @Setter генерирует геттеры и сеттеры для частных полей.

@RequiredArgsConstructor создает конструктор с требуемыми аргументами, где обязательными являются окончательные поля и поля с аннотацией *@NonNull* (подробнее об этом ниже).

```
package by.iba.model;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;
@Data
public class User {
    private int id;
    private String login;
    private String passw;
}
```

Создайте пакет *dao* (рисунок 4.41).

Рисунок 4.41 – Уровень работы с БД

```
public class UserDao {
    private Connection connection;
    public UserDao() {
        try {
            connection = ConnectorDB.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }}
    public void closeConnection() {
        try {
            if (connection != null) {
                connection.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }}
    public boolean isValidUser(final String login, final String password)
{
        PreparedStatement ps = null;
        try {
            ps = connection.prepareStatement("select login,passw from
users where login=? and passw=?");
            ps.setString(1, login);
            ps.setString(2, password);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                return true;
            }
```

```
            } catch (SQLException e) {
                e.printStackTrace();
            }
            return false;
        }
    }
```

Меняем содержимое *LoginServlet*:

```
package by.iba.servlet;
import by.iba.dao.UserDao;
import by.iba.model.ListService;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;


@WebServlet(name = "LoginServlet", urlPatterns = "/LoginServlet")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.getRequestDispatcher("/WEB-
INF/views/login.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String name = request.getParameter("name");
        String password = request.getParameter("password");

        UserDao daoUser = new UserDao();

        if (daoUser.isValidUser(name, password)) {

            request.getSession().setAttribute("name", name);
            response.sendRedirect(request.getContextPath() +
"/GroupListServlet");
// НЕТ ПАРАМЕТРОВ - всегда использует метод get
request.getRequestDispatcher("/GroupServlet")
//.forward(request, response);

        } else {
```

```
            request.setAttribute("errorMessage", "Invalid Login and
password!!");
            request.getRequestDispatcher("/WEB-INF/views/login.jsp")
                    .forward(request, response);
        }
    }

    @Override
    public void destroy() {
        super.destroy();
        System.out.println("destroy");
    }
    @Override
    public void init() throws ServletException {
        super.init();
        System.out.println("init");
    }
    @Override
    public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
        super.service(req, res);
        System.out.println("service");
    }
}
```

Если не работает, то у вас случилась проблема – не упаковывает драйвер. Добавьте в класс подключения:

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

Обязательно добавляем *cj*, если драйвер выше 8 версии.

Улучшим код, изменив следующую часть:

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

В методе *isValidUser* используется строка. Все строки лучше делать константами и размещать в начале класса.

```
package by.iba.dao;
import by.iba.util.ConnectorDB;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
public class UserDao {

    private final static String SQL_GET_USER = "select login,passw from users
where login=? and passw=?";
    private Connection connection;

    public UserDao() {
        try {
            connection = ConnectorDB.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public void closeConnection() {
        try {
            if (connection != null) {
                connection.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public boolean isValidUser(final String login, final String password) {

        PreparedStatement ps = null;
        try {
            ps = connection.prepareStatement(SQL_GET_USER);
            ps.setString(1, login);
            ps.setString(2, password);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();

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

Если будут ошибки, добавьте еще следующее:

```
db.url  = jdbc:mysql://localhost:3306/infodb?useSSL=true
```

Установление *SSL*-соединения без проверки подлинности сервера не рекомендуется.

В соответствии с требованиями *MySQL* 5.5.45+, 5.6.26+ и 5.7.6+ *SSL*-соединение должно быть установлено по умолчанию, если явный параметр не задан. Для соответствия существующим приложениям, не использующим *SSL*, свойство *verifyServerCertificate* имеет значение *false*. Вам нужно либо явно отключить *SSL*, установив *useSSL = false*, либо установить *useSSL = true* и предоставить доверительное хранилище для проверки сертификата сервера.

Еще одна проблема будет с фильтром *LoginRequiredFilter*.

Перепишем его на такой:

```
package by.iba.filter;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
@WebFilter(filterName = "LoginRequiredFilter", urlPatterns =
"/GroupListServlet")
public class LoginRequiredFilter implements Filter {
    public void destroy() {
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws ServletException, IOException {
        HttpServletRequest httpReq = (HttpServletRequest) request;
        HttpServletResponse httpResp = (HttpServletResponse) response;
        HttpSession session = httpReq.getSession();
        if (session.getAttribute("name")!=null) {
            chain.doFilter(request, response);
        }
        else {
            session.invalidate();
            request.getRequestDispatcher("LoginServlet").forward(request,
                    response);
        }
        }
        public void init (FilterConfig config) throws ServletException {
        }
    }
```

4.14 Работа с регистрацией

Допишем *Login.jsp*.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
<p><font color="red">${errorMessage}</font>
</p>
<form action="LoginServlet" method="POST">
    <p>Name : <input name="name" type="text"/>
    </p>
    <p> Password : <input name="password" type="password"/>
    </p>
    <input type="submit"/></form>
<div>
    <form action="RegisterServlet" method="GET">
        <input class ="button-main-page"   type="submit" value="Регистрация"/>
    </form>
</div>
</body>
</html>
```

Так как управление в приложении организовано через сервлеты, сделаем переход на сервлет *RegisterServlet*.

Проверим (рисунок 4.42).

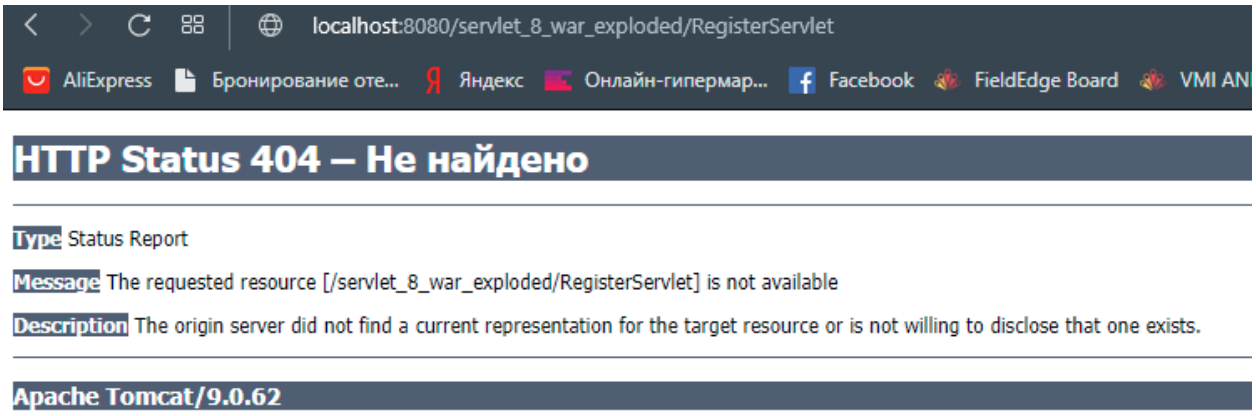


Рисунок 4.42 – Проверка недоступности *RegisterServlet*

Можно добавить к странице готовый *css*. Создайте папку *css* и разместите там стиль *style.css* (рисунок 4.43).

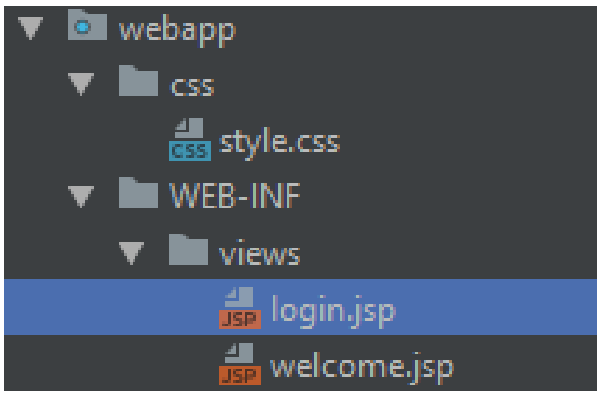


Рисунок 4.43 – Структура папок стилей

Подправим форму.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
    <title>Title</title>
</head>
<body>
<div class ="container">
    <section id="content">

        <p><font color="red">${errorMessage}</font></p>
        <form action="LoginServlet" method="POST">
            <h1> Вход в систему</h1>
            <div>
                <input placeholder="Имя" required="" id ="username" name="name"
type="text" />
            </div>
            <div>
                <input placeholder="Пароль" required=""id ="password"
name="password" type="password" />
            </div>
            <div>
                <input type="submit" value="Войти" />
            </div>
            <div>
                <a href="RegisterServlet">Регистрация</a>
            </div>
        </form>
    </section>
</div>
</body>
</html>
```

Получим применение стилей к странице входа (рисунок 4.44).

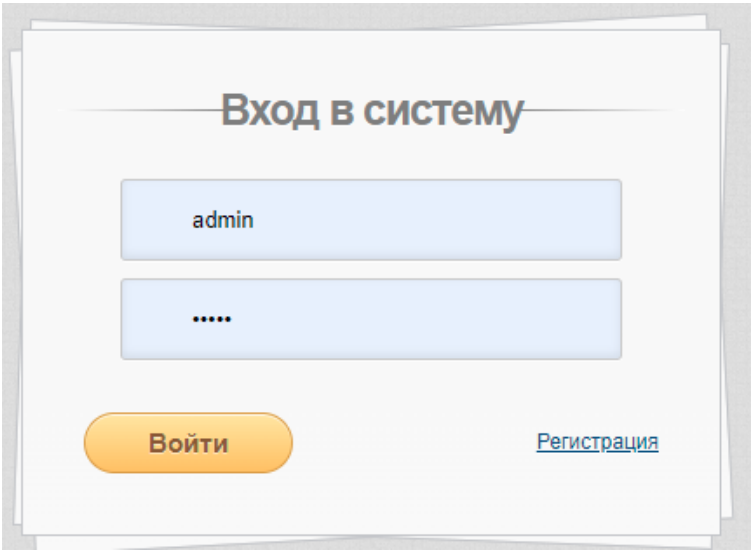


Рисунок 4.44 – Применение стилей к странице входа

Для регистрации пользователя напишите *register.jsp*, на которой пользователь должен ввести логин и пароль (рисунок 4.45).

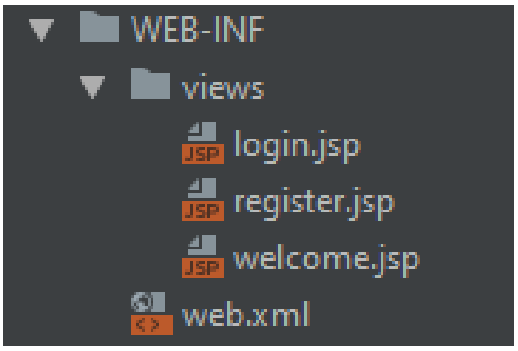


Рисунок 4.45 – Структура страниц

Добавим страницы последовательно в структуру:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
    <title>Title</title>
</head>
<body>
```

```
<div class ="container">
    <section id="content">
        <p><font color="red">${errorRegister}</font></p>

        <form action="RegisterServlet" method="POST">
            <h1> Регистрация нового пользователя </h1>
            <div>
                <input placeholder="Введите имя" required=""name="newLoginName"
type="text" />
            </div>
            <div>
                <input placeholder="Введите пароль" id ="password" required=""
name="newPassword" type="password" />
            </div>
            <div>
                <input type="submit" value="Зарегистрировать"/>
            </div>
        </form>
    </section>
</div>
</body>
</html>
```

В методе *doGet* сервлет *RegisterServlet* выполняет переход на *register.jsp*.

В методе *doPost* читаем параметры с формы, получаем соединение через *UserDao*. Потом надо добавить пользователя через вызов *daoUser.insertUser(user)*. Если метод возвращает *true*, то переходим на страницу входа. Если нет – выводим сообщение пользователю о том, что такой логин существует.

```
package by.iba.servlet;
import by.iba.dao.UserDao;
import by.iba.model.User;
@WebServlet(urlPatterns = "/RegisterServlet")
public class RegisterServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String name = request.getParameter("newLoginName");
        String password = request.getParameter("newPassword");
        UserDao daoUser = new UserDao();
        User user = new User(name, password);
        if (daoUser.insertUser(user)) {
            request.getRequestDispatcher("/WEB-
INF/views/login.jsp").forward(request, response);
        } else {
            request.setAttribute("errorRegister", "Выберите другое имя,
такой пользователь существует");
            request.getRequestDispatcher("/WEB-INF/views/register.jsp")
                    .forward(request, response); }}
```

```
      protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request
                .getRequestDispatcher("/WEB-INF/views/register.jsp")
                .forward(request, response);
    }
}
```

Допишите в класс *UserDao* метод добавления нового пользователя.

Нам понадобится запрос проверки логина и вставки нового пользователя.

Один из возможных вариантов:

```
private final static String SQL_CHECK_LOGIN = "SELECT login FROM users WHERE
login = ?";
private final static String SQL_INSERT_USER = "INSERT INTO users(login ,passw)
VALUES (? , ?)";
```

Можно написать сразу вставку с проверкой *WHERE NOT EXISTS*.

При добавлении пользователя надо проверить логин, если такой есть – метод вернет *false*, иначе добавляем:

```
public boolean insertUser(User user) {
    try {
        PreparedStatement preparedStatement =
                connection.prepareStatement(SQL_CHECK_LOGIN);
preparedStatement.setString(1, user.getLogin()); ResultSet result =
preparedStatement.executeQuery();
        if ( result.next()){ preparedStatement.close(); return false;
        }
        else {
            preparedStatement = connection.prepareStatement(SQL_INSERT_USER);

            preparedStatement.setString(1, user.getLogin());
preparedStatement.setString(2, user.getPassw());

            preparedStatement.executeUpdate(); preparedStatement.close();
        }
    } catch (SQLException e) { e.printStackTrace();
    }
    return true;
}
```

При необходимости добавьте конструктор для *User* с двумя параметрами:

```
public User(String name, String password) {
    this.login = name;
    this.passw = password;
}
```

Получим шаблон страницы проекта (рисунки 4.46–4.50).

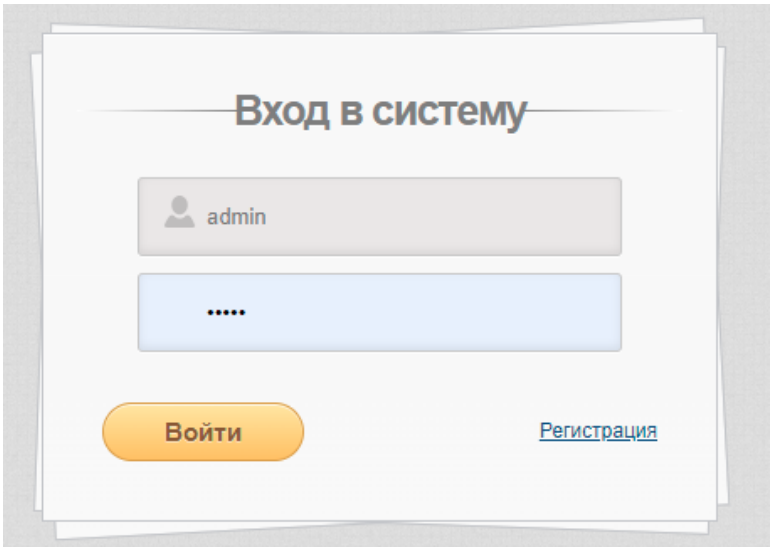


Рисунок 4.46 – Главная страница

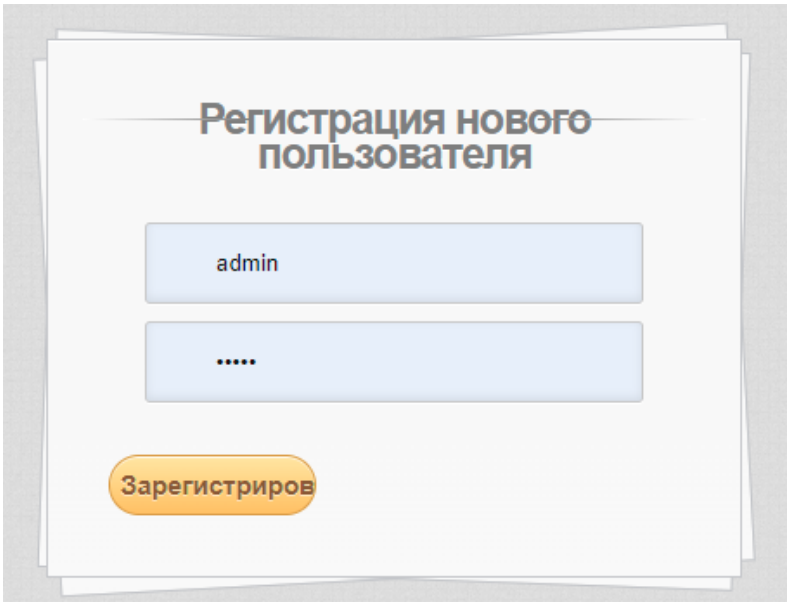


Рисунок 4.47 – Окно регистрации

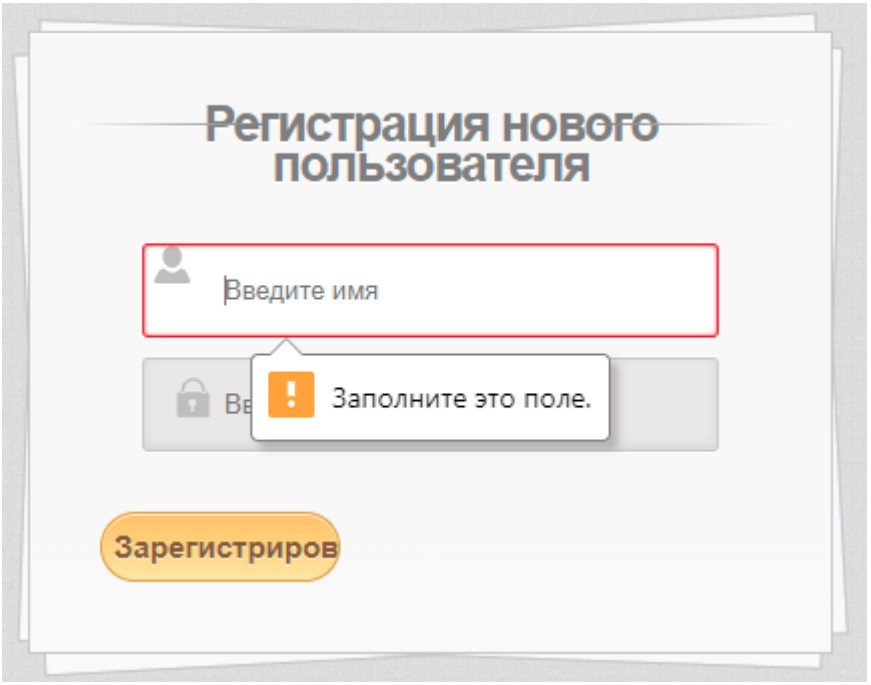


Рисунок 4.48 – Проверка регистрации и заполнения полей

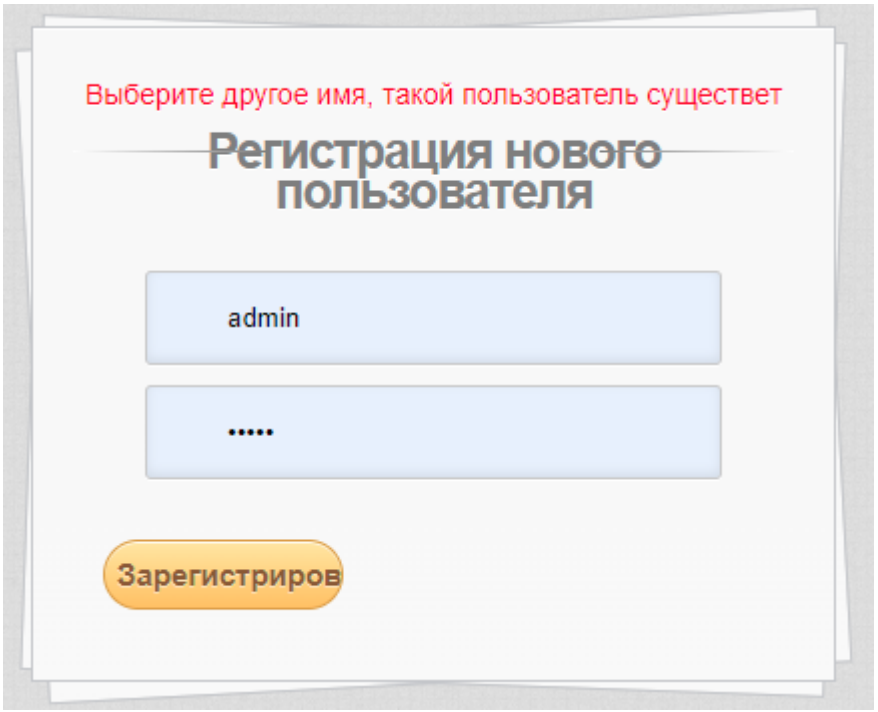


Рисунок 4.49 – Проверка ввода информации

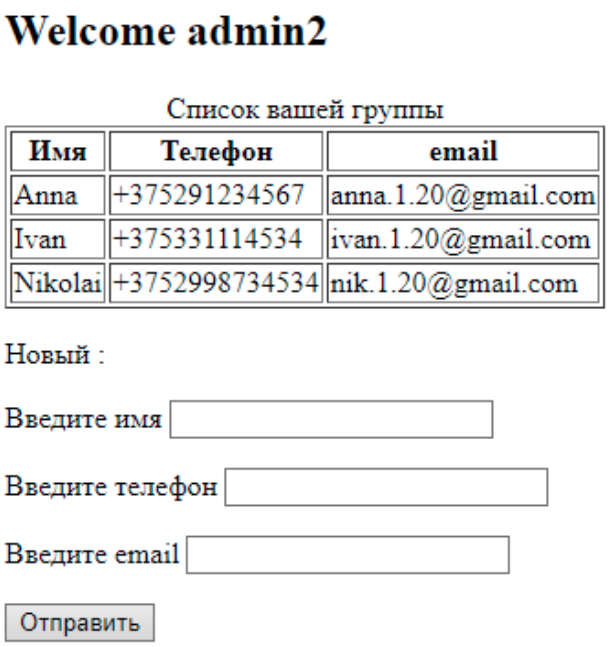


Рисунок 4.50 – Результат успешной регистрации и входа

4.15 Хеширование паролей

Hash = хеш-функция (свертка) – функция однозначного отображения строки (любой длины) на конечное множество (строку заданной длины). Само число (строка) хеш – результат вычисления хеш-функции над данными. Существуют криптографические и некриптографические хеш-функции.

В веб-приложениях хеш-функции используются для безопасного хранения паролей в базе данных. Хеширование паролей необходимо, чтобы пароли не хранились в базе в открытом виде. В случае компрометации БД реальное значение паролей может быть похищено. Если это будут хеши, то для получения реальных паролей эти хеши нужно еще взломать.

Хеш в базе должен удовлетворять следующим условиям:

1) стойкость к атакам перебора (прямой перебор и перебор по словарю);

2) невозможность поиска одинаковых паролей разных пользователей по хешам.

Для выполнения первого требования нужно использовать стойкие в настоящее время хеш-функции. Нужно выбирать более стойкий к взлому алгоритм (*SHA2* вместо *SHA1* или *MD5* и др.).

Для выполнения второго – к паролю перед хешированием добавляется случайная строка (соль). Таким образом, у двух пользователей с паролем «123456» будут разные соли. И соль, и сам хеш хранятся в базе данных.

Сделаем упрощенный вариант. Добавьте класс *HashPassword* в пакет *util*.

В классе нам будет нужен один статический метод *getHash* для хеширования строки.

Класс *MessageDigest* предоставляет приложениям функциональность алгоритма дайджеста сообщений, такого как *SHA-1*, *MD5* или *SHA-256*.

Дайджесты – это безопасные однонаправленные хеш-функции, которые принимают данные произвольного размера и выводят хеш-значение фиксированной длины.

Внесем изменения. Сначала поменяем тип пароля в базе данных. Зайдите в *Workbench* или *Heidi* и измените тип на *blob* (рисунок 4.51).

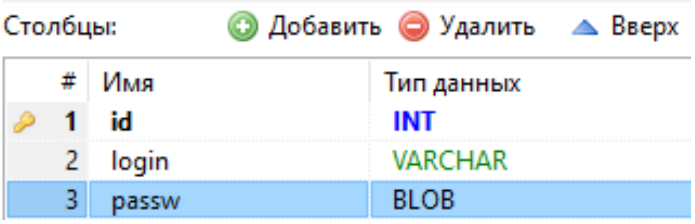


Рисунок 4.51 – Изменение типа пароля для хеширования

Отключите разрешение использования нулевых значений в поле, как представлено ниже (рисунок 4.52).

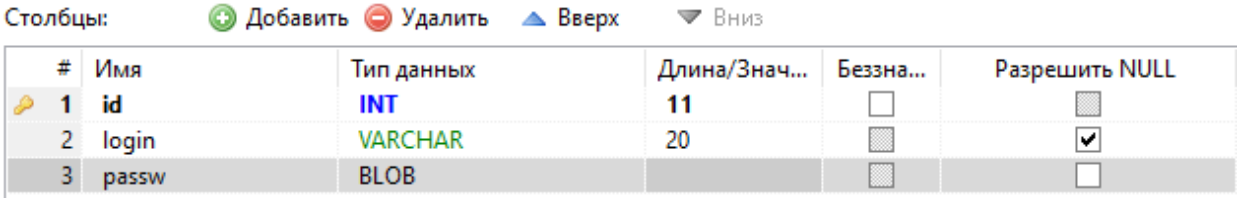


Рисунок 4.52 – Настройка полей пароля пользователя

Зайдите в класс *User* и поменяйте тип для пароля на *byte*:

```
package by.iba.model;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;
```

```
@Data
public class User {

    private int id;
    private String login;
    private byte[] passw;

    public User(String name, byte[] password) {
        this.login = name;
        this.passw = password;
    }
}
```

Также в *UserDao*.
Методы *isValid* и *insert* будут содержать *byte*-данные.

```
package by.iba.dao;
import by.iba.model.User;
import by.iba.util.ConnectorDB;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserDao {
    private final static String SQL_GET_USER = "select login,passw from
users where login=? and passw=?";
    private final static String SQL_CHECK_LOGIN = "SELECT login FROM users
WHERE login = ?";
    private final static String SQL_INSERT_USER = "INSERT INTO users(login
,passw) VALUES (? , ?)";
    private Connection connection;
    public UserDao() {
        try {
            connection = ConnectorDB.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public void closeConnection() {
        try {
            if (connection != null) {
                connection.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

```
    public boolean isValidUser(final String login, final byte[] password)
{
        PreparedStatement ps = null;
        try {
            ps = connection.prepareStatement(SQL_GET_USER);
            ps.setString(1, login);
            ps.setBytes(2, password);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (ps != null) {
                try {
                    ps.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }}
        }
        return false;
    }
    public boolean insertUser(User user) {
        try {
            PreparedStatement preparedStatement =
                    connection.prepareStatement(SQL_CHECK_LOGIN);
preparedStatement.setString(1, user.getLogin()); ResultSet result =
preparedStatement.executeQuery();
            if ( result.next()){ preparedStatement.close(); return false;
            }
            else {
                preparedStatement =
connection.prepareStatement(SQL_INSERT_USER);
                preparedStatement.setString(1, user.getLogin());
preparedStatement.setBytes(2, user.getPassw());
                preparedStatement.executeUpdate(); preparedStatement.close();
            }
        } catch (SQLException e) { e.printStackTrace();  }
        return true;
    }
}
```

В *LoginServlet* при проверке пользователя передавать надо хеш-значение

```
    if (daoUser.isValidUser(name, HashPassword.getHash(password)))
```

В *RegisterServlet* аналогично:

```
    User user = new User(name, HashPassword.getHash(password));
```

Добавим класс *HashPassword* в пакет *util:*

```
package by.iba.util;
import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashPassword {

    public static byte[] getHash(String passStr) {
        MessageDigest digest = null;
        byte[] hash = null;
        try {
            digest = MessageDigest.getInstance("MD5");
            digest.reset();
            hash = digest.digest(passStr.getBytes("UTF-8"));

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        return hash;
    }

}
```

Очистите БД и пересоздайте запись через проект (рисунок 4.53).

infodb.users: 1 строк (приблизительно)

id	login	passw
4	admin	!#/)zWГ§C‰J□ЈЬГ

Рисунок 4.53 – Проверка шифрования пароля

В зависимости от кодировки увидите такую ситуацию.

Управление списком студентов

Так же как и для пользователей *User*, надо добавить хранение, извлечение и добавление персон в БД. Для этого добавим новую таблицу в базу данных (рисунок 4.54).

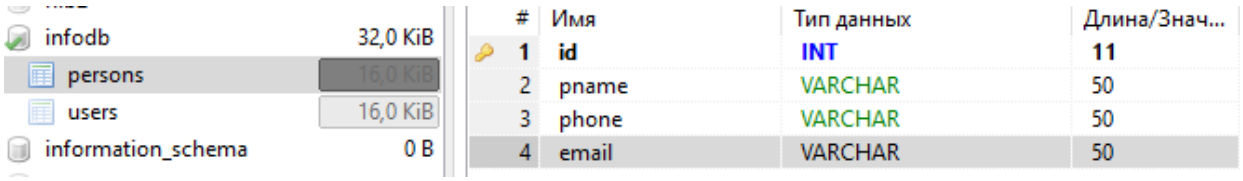


Рисунок 4.54 – Структура сущностей БД

Создадим *PersonDao* (рисунок 4.55).

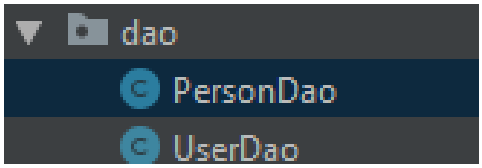


Рисунок 4.55 – Модификация объектной модели и уровня работы с БД

```
package by.iba.dao;
import by.iba.model.Person;
import by.iba.util.ConnectorDB;
import java.sql.*;
import java.util.LinkedList;
import java.util.List;

public class PersonDao {
    private final static String SQL_GET_PERSONS = "SELECT * FROM persons";
    private final static String SQL_INSERT_PERSONS = "INSERT INTO persons(pname,
phone, email) VALUES (?,?,?)";
    private static Connection connection;
    public PersonDao() {
        try {
            if (connection == null) {
                connection = ConnectorDB.getConnection();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public void closeConnection() {
        try {
            if (connection != null) {
                connection.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } }
```

```
 public void insertPerson(Person person) {
        try {
            PreparedStatement preparedStatement =
                    connection.prepareStatement(SQL_INSERT_PERSONS);
            preparedStatement.setString(1, person.getName());
            preparedStatement.setString(2, person.getPhone());
            preparedStatement.setString(3, person.getEmail());
            preparedStatement.executeUpdate();
            preparedStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    public List<Person> getPersons() {
        List<Person> persons = new LinkedList<Person>();
        try {
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(SQL_GET_PERSONS);
            Person person = null;
            while (resultSet.next()) {
                person = new Person();
                person.setName(resultSet.getString("pname"));
                person.setPhone(resultSet.getString("phone"));
                person.setEmail(resultSet.getString("email"));
                persons.add(person);
            }
            resultSet.close();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return persons;
    }}
```

Перепишите *GroupListServlet* для работы с *PersonDao*.

Обязательно добавляем новую зависимость в *pom.xml*.

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.11</version>
</dependency>
```

Модифицируем класс *GroupListServlet* для обработки запросов к БД.

```
package by.iba.servlet;
import by.iba.dao.PersonDao;
import by.iba.model.ListService;
import by.iba.model.Person;
```

```
@WebServlet(name = "GroupListServlet", value = "/GroupListServlet")
public class GroupListServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        PersonDao daoPerson = new PersonDao();
        String nname = request.getParameter("nname");
        String nphone = request.getParameter("nphone");
        String nemail = request.getParameter("nemail");
        if (nname.isEmpty() || nphone.isEmpty() || nemail.isEmpty()) {
            request.setAttribute("errorMessage", "Заполните все поля");
        } else {
            daoPerson.insertPerson(new Person(nname, nphone, nemail));
        }
        request.setAttribute("group", daoPerson.getPersons());
        request.getRequestDispatcher("/WEB-
INF/views/welcome.jsp").forward(request, response);
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        PersonDao daoPerson = new PersonDao();
        request.setAttribute("group", daoPerson.getPersons());
        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp")
                .forward(request, response);
    }
}
```

Проверяем (рисунок 4.56).

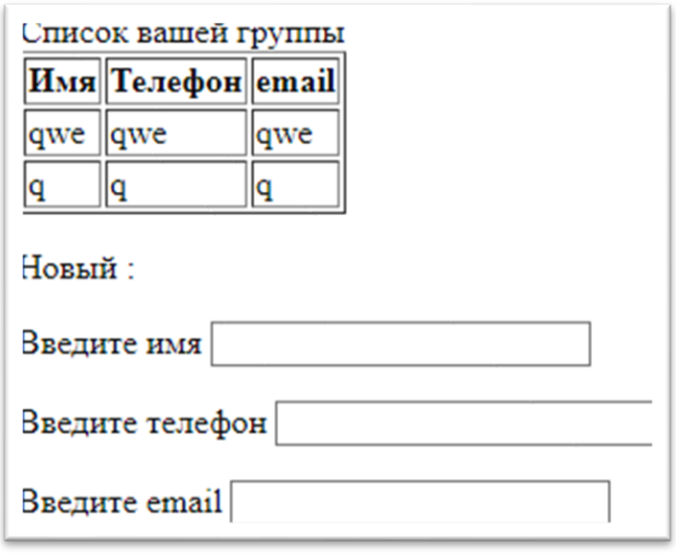


Рисунок 4.56 – Результат вывода информации из БД

4.16 Добавление поддержки *Cookie*

Для хранения информации на компьютере клиента используются возможности класса *javax.servlet.http.Cookie*.

Cookie – это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах *Cookies*. Клиент может запретить браузеру прием файлов *Cookies*. Браузер возвращает информацию обратно на сервер как часть заголовка *HTTP*, когда клиент повторно заходит на тот же веб-ресурс.

Cookies могут быть ассоциированы не только с сервером, но и также с доменом, в этом случае браузер посылает их на все серверы указанного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (*Single Signon*), где серверы одного домена обмениваются идентификационными маркерами (*token*) с помощью общих *Cookies*. *Cookie* были созданы в компании *Netscape* как средства отладки, но теперь используются повсеместно.

Файл *Cookie* – это файл небольшого размера для хранения информации, который создается серверным приложением и размещается на компьютере пользователя. Браузеры накладывают ограничения на размер файла *Cookie* и общее количество *Cookie*, которые могут быть установлены на пользовательском компьютере приложениями одного веб-сервера. Чтобы послать *Cookie* клиенту, сервлет должен создать объект класса *Cookie*, указав конструктору имя и значение блока, и добавить их в объект *response*. Конструктор использует имя блока в качестве первого параметра, а его значение – в качестве второго.

```
Cookie cookie = new Cookie("model", "Canon D7000"); response.addCookie (cookie);
```

Извлечь информацию *Cookie* из запроса можно с помощью метода *getCookies()* объекта *HttpServletRequest*, который возвращает массив объектов, составляющих этот файл.

```
Cookie[ ] cookies = request.getCookies();
```

После этого для каждого объекта класса *Cookie* можно вызвать метод *getValue()*, который возвращает строку *String* с содержимым блока *Cookie*. Экземпляр *Cookie* имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из ключевых является срок жизни в секундах от момента первой отправки клиенту, определить который следует методом

setMaxAge(long sec). Если параметр не указан, то *Cookie* существует только до момента прерывания контакта клиента с приложением.

Добавим работу с файлом *Cookie*, который хранит последнее время обращения на сервер *(name time)*. При авторизации пользователя выводится приветствие и строка с временем последнего входа.

Сервлет *LoginServlet* добавляет *Cookie* к объекту-ответа:

```
package by.iba.servlet;
import by.iba.dao.UserDao;
import by.iba.model.ListService;
import by.iba.util.HashPassword;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;

@WebServlet(name = "LoginServlet", urlPatterns = "/LoginServlet")
public class LoginServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.getRequestDispatcher("/WEB-
INF/views/login.jsp").forward(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String name = request.getParameter("name");
        String password = request.getParameter("password");

        UserDao daoUser = new UserDao();
        if (daoUser.isValidUser(name, HashPassword.getHash(password))) {
            request.getSession().setAttribute("name", name);
            Cookie[] cookies = request.getCookies();
            if (cookies != null) {
                for (Cookie c : cookies) { Cookie cookie = c;
                    System.out.println(cookie.getName() + cookie.getValue());
                    if (name.equals(cookie.getName())) {
```

```
                        request. getSession().setAttribute("lastdate",
                                cookie.getValue());
                    }
                }
            }
            Cookie userCookie = new Cookie(name,
LocalDateTime.now().toString());
            userCookie.setMaxAge(60 * 60 * 24 * 365); //хранить куки год
            response.addCookie(userCookie);
            response.sendRedirect(request.getContextPath() +
"/GroupListServlet");
request.getRequestDispatcher("/GroupServlet")
        } else {
            request.setAttribute("errorMessage", "Invalid Login and
password!!");
            request.getRequestDispatcher("/WEB-INF/views/login.jsp")
                    .forward(request, response);
        }
    }
    @Override
    public void destroy() {
        super.destroy();
        System.out.println("destroy");
    }
    @Override
    public void init() throws ServletException {
        super.init();
        System.out.println("init");
    }
    @Override
    public void service(ServletRequest req, ServletResponse res) throws
ServletException, IOException {
        super.service(req, res);
        System.out.println("service");
    }
}
```

Чтобы посмотреть, что вы сохранили в браузере, в данном случае в *Chrome*:

1) Нажмите значок с тремя точками в правом верхнем углу экрана «Настройки».

2) Внизу выберите пункт меню «Дополнительные».

3) В разделе «Конфиденциальность и безопасность» нажмите пункт «Настройки контента».

4) Выберите файлы *Cookie* и найдите окно «*localhost»* (рисунок 4.57).

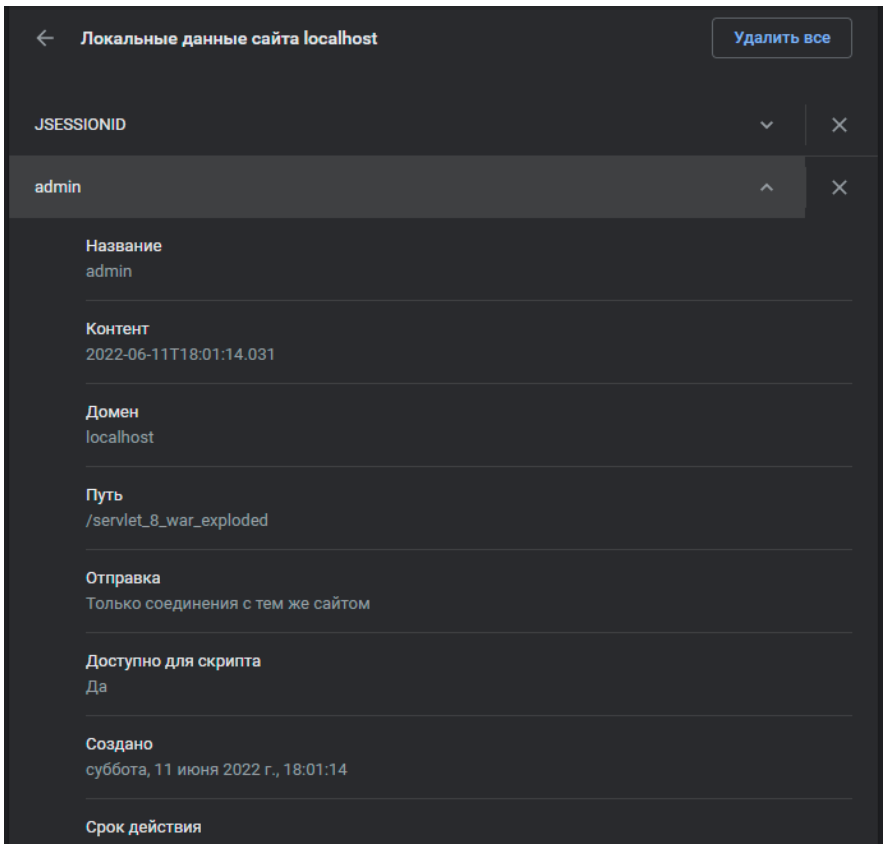


Рисунок 4.57 – Проверка сохранения *Cookies*

Добавьте *CSS* (рисунок 4.58).

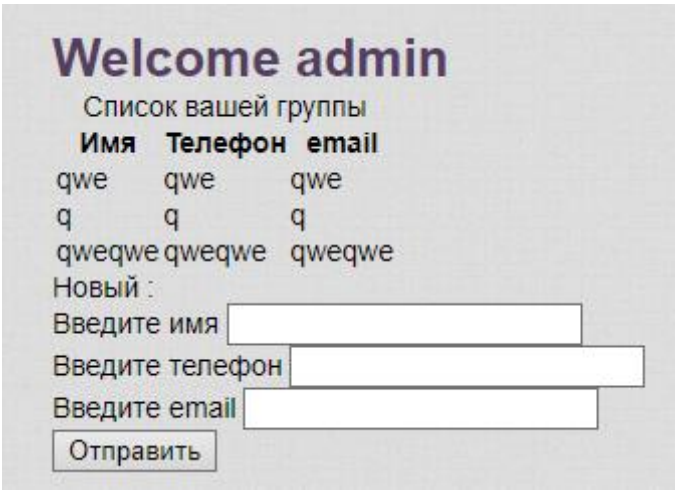


Рисунок 4.58 – Применение стиля к форме страницы

Java-программы – это очень часто большие серверные приложения без *UI*, консоли и т. д. Они обрабатывают одновременно запросы тысяч пользователей, и нередко при этом возникают различные ошибки. Особенно, когда разные нити начинают друг другу мешать.

Фактически, единственным способом поиска редко воспроизводимых ошибок и сбоев в такой ситуации является запись в лог-файл всего, что происходит в каждой нити.

Чаще всего в лог пишется информация о параметрах метода, с которыми он был вызван, все перехваченные ошибки и еще много промежуточной информации.

Чем полнее лог, тем легче восстановить последовательность событий и отследить причины возникновения сбоя или ошибки.

Логирование – это возможность следить за процессом выполнения бизнес-логики проекта. Вы можете выводить нужные вам логи в отдельный файл. Для этого нужно реализовать свою систему логирования, где вы сможете указать, какие из логов необходимо выводить.

Изначально в *Java* не было своего логгера, что привело к написанию нескольких независимых логгеров. В *Java* для этого используется:

– *JUL – java.util.logging*.
– *log4j*.
– *JCL – jakarta commons logging*.
– *Logback*.
– *SLF4J – simple logging facade for java*.

Самым распространенным из них стал *Log4j*. Будем использовать *Log4j*. Данный фреймворк на текущий момент имеет уже вторую версию, которая не совместима с первой. Для использования *Log4j2* вам необходимо подключить библиотеки *log4j-api-2.x* и *log4j-core-2.x*.

Найдите в Maven-репозитории и добавьте к проекту новые зависимости.

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.14.1</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.14.1</version>
</dependency>
```

Чтобы гибко управлять логированием, создайте в папке *resources*/ файл *log4j2.properties* (рисунок 4.59).

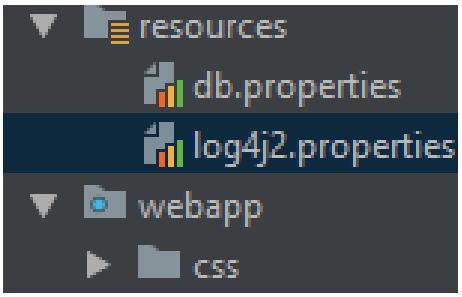


Рисунок 4.59 – Файл настройки логирования

Поскольку разные приложения предъявляют разные требования к ведению журнала, вы можете настроить *Log4j2* соответствующим образом. Кроме того, вам часто придется изменять конфигурации *Log4j2* приложения на протяжении всего жизненного цикла развертывания.

Во время локальной разработки вы можете работать с консольным приложением, чтобы избежать накладных расходов на ввод-вывод файлов, а в других средах развертывания установить файловый аппендер или какое-либо другое постоянное место назначения для сохранения сообщений журнала.

Настроить *Log4J2* можно программно или через файлы конфигурации, такие как свойства, *XML*, *JSON* и *YAML*, находящиеся в пути к классам вашего проекта. Благодаря использованию файлов конфигурации, у вас есть возможность изменять различные параметры конфигурации без изменения кода вашего приложения.

В этом проекте у нас будет файл свойств. *log4j2.properties* представляет собой набор пар «ключ:значение» с параметрами для настройки различных компонентов, таких как регистраторы, приложения и макеты.

```
name=PropertiesConfig
property.filename = logs
appenders = console, file
appender.console.type=Console
appender.console.name = STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L -
%m%n
appender.file.type=File
appender.file.name = LOGFILE
appender.file.fileName=${filename}/appservlet.log
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
```

```
%c{1} -
%msg%n
loggers=file
logger.file.name=by.iba
logger.file.level = debug
logger.file.appenderRefs = file
logger.file.appenderRef.file.ref=LOGFILE
rootLogger.level=debug
rootLogger.appenderRefs = stdout
rootLogger.appenderRef.stdout.ref = STDOUT
```

Обычно у каждого лог-сообщения есть своя степень важности и, используя ее, можно часть этих сообщений отбрасывать.

Степени важности (рисунок 4.60).

Степень важности	Описание
ALL	Все сообщения
TRACE	Мелкое сообщение при отладке
DEBUG	**Сообщения важные при отладке**
INFO	**Просто сообщение**
WARN	**Предупреждение**
ERROR	**Ошибка**
FATAL	**Фатальная ошибка**
OFF	Нет сообщения

Рисунок 4.60 – Степени важности и очередность логирования

Если выставить уровень логирования в *WARN*, то все сообщения, менее важные, чем *WARN*, будут отброшены: *TRACE*, *DEBUG*, *INFO*.

Если выставить уровень фильтрации в *FATAL*, то будут отброшены даже *ERROR*.

Есть еще два уровня важности, которые используются при фильтрации – это *OFF* – отбросить все сообщения и *ALL* – показать все сообщения (не отбрасывать ничего).

Добавим:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

и сам класс:

```
public class ConnectorDB {
    private static final Logger logger =LogManager.getLogger(ConnectorDB.class);
    public static Connection getConnection() throws SQLException {
        DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
        ResourceBundle resource = ResourceBundle.getBundle("db",
Locale.getDefault());
        String url = resource.getString("db.url");
        String user = resource.getString("db.user");
        String pass = resource.getString("db.password");
        logger.info("connection establish");
        return DriverManager.getConnection(url, user, pass);
    }}
```

Результат создания лога запуска приложения с применением атрибутов конфигурации представлен в консоли (рисунки 4.61 и 4.62).

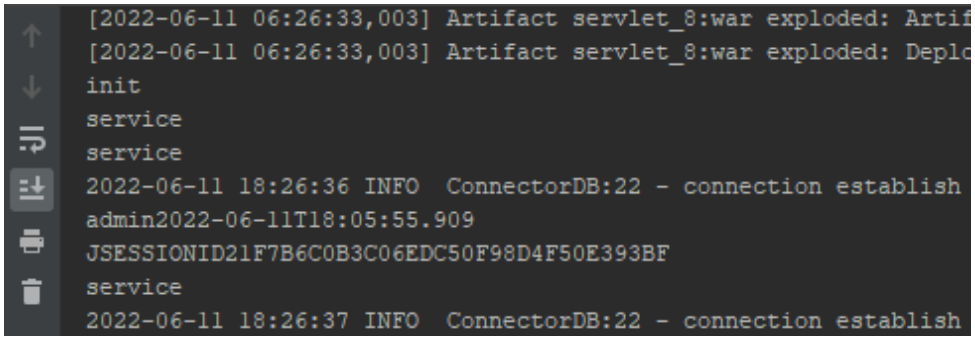


Рисунок 4.61 – Результат логирования в консоли

Результат логирования информации в файл представлен на рисунке 4.61.

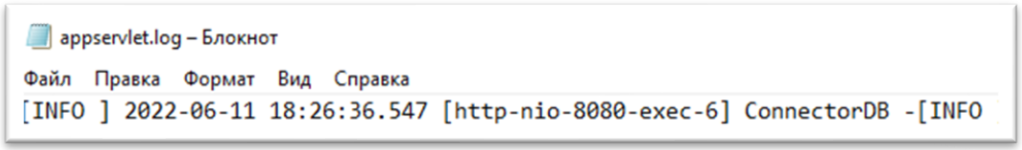


Рисунок 4.62 – Результат логирования в файле

Добавим логирование в *PersonDao*:

```
package by.iba.dao;
import org.apache.logging.log4j.LogManager;
import by.iba.model.Person;
import by.iba.util.ConnectorDB;
import java.sql.*;
import java.util.LinkedList;
import java.util.List;

public class PersonDao {
    private static final Logger logger =
LogManager.getLogger(PersonDao.class);
    private final static String SQL_GET_PERSONS = "SELECT * FROM persons";
    private final static String SQL_INSERT_PERSONS = "INSERT INTO
persons(pname, phone, email) VALUES (?,?,?)";
    private static Connection connection;
    public PersonDao() {
        try {
            if (connection == null) {
                connection = ConnectorDB.getConnection();
                logger.info("get connection");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }}
    public void closeConnection() {
        try {
            if (connection != null) {
                connection.close();
                logger.info("close connection");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void insertPerson(Person person) {
        try {
            PreparedStatement preparedStatement =
                   connection.prepareStatement(SQL_INSERT_PERSONS);
            preparedStatement.setString(1, person.getName());
            preparedStatement.setString(2, person.getPhone());
            preparedStatement.setString(3, person.getEmail());
            preparedStatement.executeUpdate();
            preparedStatement.close();
            logger.info("New Person " + person.getName() +" inserted");
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
```

```
public List<Person> getPersons() {
    List<Person> persons = new LinkedList<Person>();
    try {
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(SQL_GET_PERSONS);
        Person person = null;
        while (resultSet.next()) {
            person = new Person();
            person.setName(resultSet.getString("pname"));
            person.setPhone(resultSet.getString("phone"));
            person.setEmail(resultSet.getString("email"));
            persons.add(person);
        }
        resultSet.close();
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return persons;
}}
```

Тема 5. Основы *Spring Framework*

5.1 Конфигурация проекта

Для начала необходимо создать проект в среде разработки (*IntelliJ* (желательно), *NetBeans*, *Eclipse*). *File->New->Project.*

Переходим в *Maven* и выбираем архетип (рисунки 5.1–5.4).

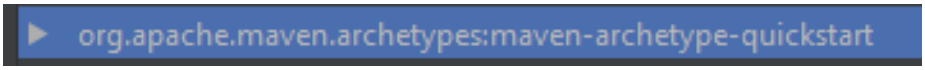


Рисунок 5.1 – Выбор архетипа

Далее задаем *GroupId* с указанием доменной зоны и фамилии через точку. В поле *ArtifactId* задаем имя проекта. *Version* оставляем как есть (рисунок 5.2).

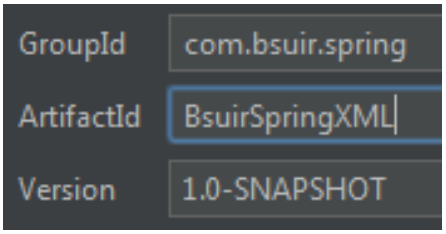


Рисунок 5.2 – Инициализация проекта

Проверяем правильность внесенных данных (рисунок 5.3).

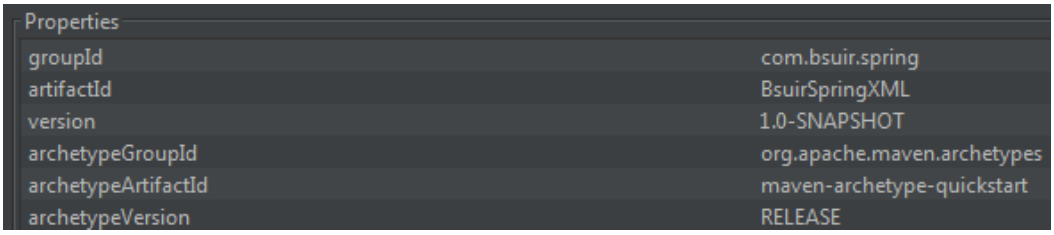


Рисунок 5.3 – Проверка правильности конфигурации проекта

Задаем имя проекту (рисунок 5.4).

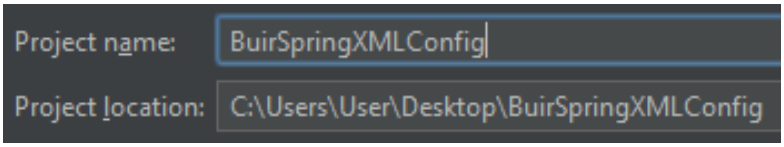


Рисунок 5.4 – Указание имени проекта

Для начала несколько слов о том, что же такое *Spring*. В настоящее время под термином *Spring* часто подразумевают целое семейство проектов. В большинстве своем они развиваются и курируются компанией *Pivotal* и силами сообщества. Ключевые (но не все) проекты семейства *Spring* представлены ниже.

Spring Framework (или *Spring Core*)

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

Spring MVC (часть *Spring Framework*)

Spring оперирует понятиями контроллеров, маппингов запросов, различными *HTTP*-абстракциями и т. п. Со *Spring MVC* интегрированы нормальные шаблонные движки, типа *Thymeleaf, Freemaker, Mustache*, а также есть сторонние интеграции с множеством других.

Spring Data

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

Spring Cloud

Много полезного для микросервисной архитектуры – *service discovery*, трассировка и диагностика, балансировщики запросов, *circuit breaker*, роутеры и т. п.

Spring Security

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

Spring Integration

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

Плюсы *Spring*:

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

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

– *Spring* является модульной средой и позволяет использовать отдельные свои части без необходимости вводить остальные;

– возможность работы с *POJO* (без контейнеров *EJB*);

– существует большое количество расширений *Spring* для построения приложений на *Java Enterprise*-платформе;

– сообщество *Spring* – одно из лучших сообществ из всех проектов с открытым исходным кодом, списки рассылки и форумы всегда активны;

– у *Spring* отличная подробная документация.

Spring обеспечивает решение многих задач, с которыми сталкиваются *Java*-разработчики и организации, которые хотят создать информационную систему, основанную на платформе *Java*. Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. *Spring* не всецело связан с платформой *Java Enterprise*, несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности.

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

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

Возможности *Spring*:

– использование внедрения зависимостей (*DI*);

– поддержка аспектно-ориентированного программирования (в том числе интеграция с *AspectJ*);

– язык выражений *Spring* (*SpEL*) – позволяет приложению манипулировать объектами *Java* во время выполнения;

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

– *Spring* обеспечивает хорошую интеграцию с большинством инструментов доступа к данным (*JDBC, Hibernate, MyBatis, JDO, JPA* и т. п.);

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

– интеграция с *JEE* – внедрение бинов *Spring* в компоненты *EJB*;

– *MVC* на веб-уровне;

– поддержка электронной почты;

– поддержка планирования заданий;

– поддержка динамических сценариев (*Groovy, JRuby, BeanShell*).

Spring Annotation

Вначале был *EJB 2.1* с его огромным количеством *XML*-файлов. Не будет особым преувеличением, если сказать, что на одну строку кода для бизнес-логики нужно было написать по крайней мере 10 строк кода от фреймворка и две страницы *XML*, содержащие локальные и удаленные интерфейсы, ручной *JNDI-lookup*, многоуровневые *try-catch*, проверки на *RemoteException*. Потом создали *Spring Framework*.

Прошло время и *Sun* провела работу над ошибками. *EJB 3.0* был даже проще *Spring, XML-free,* с аннотациями, *dependency injection, EJB 3.1* стал еще одним огромным шагом в сторону упрощения.

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

Уход от *XML* рассмотрим во втором примере теоретической части. Существует множество аннотаций:

@Component – аннотация для любого компонента фреймворка;

@Service (сервис-слой приложения) – аннотация, объявляющая, что этот класс представляет собой сервис – компонент сервис-слоя. Сервис является подтипом класса *@Component*. Использование данной аннотации позволит искать бины-сервисы автоматически;

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

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

@ResponseBody – аннотация показывает, что данный метод может возвращать кастомный объект в виде *xml, json*;

@RestController – аннотация аккумулирует поведение двух аннотаций *@Controller* и *@ResponseBody*;

@Transactional – перед выполнением метода, помеченного данной аннотацией, начинается транзакция, после выполнения метода осуществляется коммит транзакции, а при выбрасывании *RuntimeException* – ее возвращение в первоначальное состояние;

@Autowired – аннотация позволяет автоматически установить значение поля;

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

@ModelAttribute – аннотация, связывающая параметр метода или возвращаемое значение метода с атрибутом модели, которая будет использоваться при выводе *JSP*-страницы;

@PathVariable – аннотация, которая показывает, что параметр метода должен быть связан с переменной из *URL*-адреса;

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

@PostConstruct – аннотация для метода, который будет вызван после вызова конструктора бина;

@PreDestroy – аннотация для метода, который будет вызван перед уничтожением бина;

@Profile – аннотация для создания профилей конфигурации проекта. Может применяться как к бинам, так и к конфигурационным классам.

5.2 Основы *Spring Core*

Рассмотрим приложение, которое выводит в консоль строки, состоящие из букв и цифр, размещенных в рандомном порядке (количество строк, количество символов в строке и интервалы выбирает пользователь).

Структура проекта представлена ниже (рисунок 5.5).

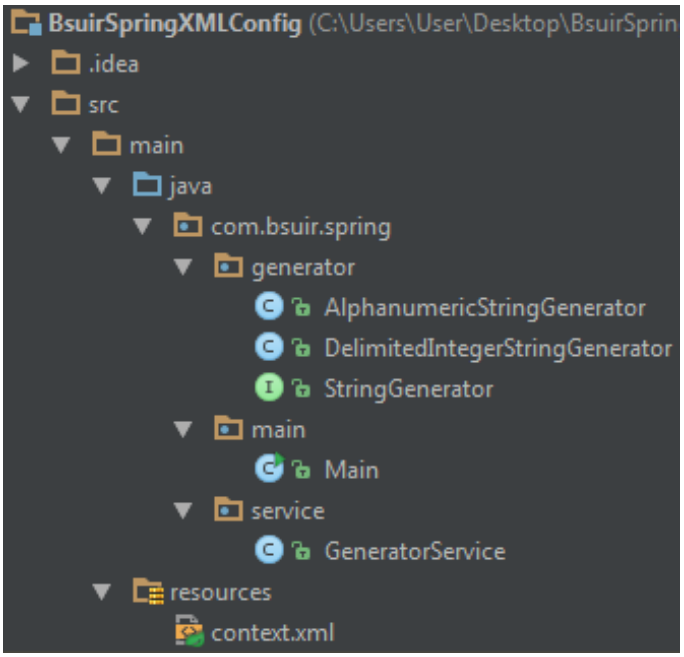


Рисунок 5.5 – Структура проекта

В файле *pom.xml* нужно подключить зависимости, необходимые для работы со *Spring* (*spring-core* и *spring-context*):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.bsuir.spring</groupId>
    <artifactId>BsuirSpringXML</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
    <properties>
        <spring.version>4.3.9.RELEASE</spring.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

Создадим интерфейс *StringGenerator* единственным методом, который осуществляет генерацию строки.

```
package com.bsuir.spring.generator;
public interface StringGenerator {
    String EMPTY_STRING = "";
    String generate(int count);
}
```

Реализуем данный интерфейс двумя способами.

С генерацией строки букв:

```
package com.bsuir.spring.generator;
import java.util.Random;
public class AlphanumericStringGenerator implements StringGenerator {
    private static final Random RANDOM = new Random();
    private String source;
    public AlphanumericStringGenerator(String source) {
        this.source = source;
    }
    @Override
    public String generate(int count) {
        if (source == null || source.isEmpty()) {
            return EMPTY_STRING;
        }
        if (count < 0) {
            throw new IllegalArgumentException("Length of generated string must
be positive.");
        }
```

```
        StringBuilder result = new StringBuilder();
        for (int index = 0; index < count; index++) {
            int sourceIndex = RANDOM.nextInt(source.length());
            result.append(source.charAt(sourceIndex));
        }
        return result.toString(); }
}
```

С генерацией строки цифр:

```
package com.bsuir.spring.generator;
import java.util.Random;
public class DelimitedIntegerStringGenerator implements StringGenerator {

    private static final Random RANDOM = new Random();
    private String delimiter;
    private int endInclusive;

    public DelimitedIntegerStringGenerator(String delimiter, int endInclusive) {
        this.delimiter = delimiter;
        this.endInclusive = endInclusive;
    }

    @Override
    public String generate(int count) {
        if (endInclusive <= 0) {
            throw new IllegalArgumentException("Property endInclusive must be
positive integer.");
        }
        if (count < 0) {
            throw new IllegalArgumentException("Length of generated string must
be positive.");
        }
        StringBuilder result = new StringBuilder();
        for (int index = 0; index < count; index++) {
            result.append(RANDOM.nextInt(endInclusive)).append(delimiter);
        }
        return result.deleteCharAt(count - 1).toString();
    }
}
```

Создадим класс-сервис, осуществляющий генерацию текста и цифр, а также содержащий геттеры и сеттеры.

```
package com.bsuir.spring.service;
import com.bsuir.spring.generator.StringGenerator;
import java.util.ArrayList;
import java.util.List;
```

```
public class GeneratorService {

    private StringGenerator stringGenerator;
    private int stringsCount;
    private int length;
    public GeneratorService() {
    }

    public GeneratorService(StringGenerator stringGenerator, int stringsCount,
int length) {
        this.stringGenerator = stringGenerator;
        this.stringsCount = stringsCount;
        this.length = length;
    }

    public List<String> generateStrings() {
        List<String> strings = new ArrayList<>();
        for (int count = 0; count < stringsCount; count++) {
            strings.add(stringGenerator.generate(length));
        }
        return strings;
    }

    public StringGenerator getStringGenerator() {
        return stringGenerator;
    }

    public int getStringsCount() {
        return stringsCount;
    }

    public void setStringGenerator(StringGenerator stringGenerator) {
        this.stringGenerator = stringGenerator;
    }

    public void setStringsCount(int stringsCount) {
        this.stringsCount = stringsCount;
    }

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }
}
```

Создадим бины, соответствующие данным классам, и сконфигурируем их. Опишем конфигурацию в файле *string.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <!--Создание бина для класса AlphanumericStringGenerator.-->
    <bean id="alphanumericStringGenerator"
class="com.bsuir.spring.generator.AlphanumericStringGenerator">
        <!--Внедрение значения "append" типа String как аргумент конструктора-->
        <constructor-arg type="java.lang.String"
value="abcdefghijkABCDEFGHIJK123467890" />
    </bean>

    <!--Создание бина для класса DelimitedIntegerStringGenerator.-->
    <bean id="delimitedIntegerStringGenerator"
class="com.bsuir.spring.generator.DelimitedIntegerStringGenerator">
        <!--Внедрение полей класса при помощи сеттеров-->
        <constructor-arg type="java.lang.String" value="," />
        <constructor-arg type="int" value="10" />
    </bean>

    <!--Создание бина для класса GeneratorService.-->
    <bean id="generatorService"
class="com.bsuir.spring.service.GeneratorService">
        <!--Внедрение бина с id "alphanumericStringGenerator" как поле
stringGenerator с использованием сеттера-->
        <property name="stringGenerator" ref="delimitedIntegerStringGenerator"/>
        <!--Внедрение значения 20 в поле "text" с использованием сеттера-->
        <property name="stringsCount" value="1" />
        <property name="length" value="5" />
    </bean>

</beans>
```

Внедрять свойства через конструктор можно тегом *<constructor-arg/>*. При использовании этого метода внедрения в классе, для которого создается бин, происходит вызов конструктора с параметрами тех типов, которые заданы в атрибуте *type*. Так, например, при генерировании первого бина будет вызван конструктор *AlphanumericStringGenerator()*, куда будет передано значение «*abcdefghijkABCDEFGHIJK123467890*». Отметим, что тегов *<constructor-arg />* для одного бина может быть объявлено несколько: произойдет вызов конструктора с несколькими параметрами. В то же время, если ни одного тега *<constructor-arg />* не задано, будет вызван конструктор без параметров (как, например, в третьем бине).

При внедрении значений через свойства используется тег <*property* />, где определяется название поля для внедрения и значение, которое необходимо передать в метод *set*, который должен иметь атрибут доступа *public*. Так в бине с *GeneratorService* вначале с использованием конструктора без параметров будет создан экземпляр класса *GeneratorService*, а затем будут вызваны методы *setStringGenerator*, куда будет передана ссылка на бин с *delimitedIntegerStringGenerator*, и методы *setStringsCount* со значением параметра 1 и *setLength* со значением 5.

Также заметим, что для внедрения стандартных типов можно использовать атрибут *value*, куда передается строковое значение, в дальнейшем преобразуемое в объект необходимого типа. Если же необходимо внедрить ссылку на бин контекста, нужно использовать атрибут *ref*, куда передать идентификатор бина.

Еще одним способом внедрения зависимостей является использование фабричного метода. Если данный метод является статическим, то необходимо в атрибуте *class* указать имя класса, содержащего данный метод, а в атрибуте *factory-method* указать имя метода.

Создадим класс *Main* с точкой входа.

```
package com.bsuir.spring.main;
import com.bsuir.spring.service.GeneratorService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import java.util.List;

public class Main {

    private static final String CONFIGURATION_PATH = "context.xml";
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext(CONFIGURATION_PATH);
        GeneratorService service = (GeneratorService)
context.getBean("generatorService");
        List<String> generated = service.generateStrings();
        for (String s : generated) {
            System.out.println(s);
        }
    }
}
```

Результат выполнения кода со значением бина id="delimitedIntegerStringGenerator" представлен на рисунке 5.6.

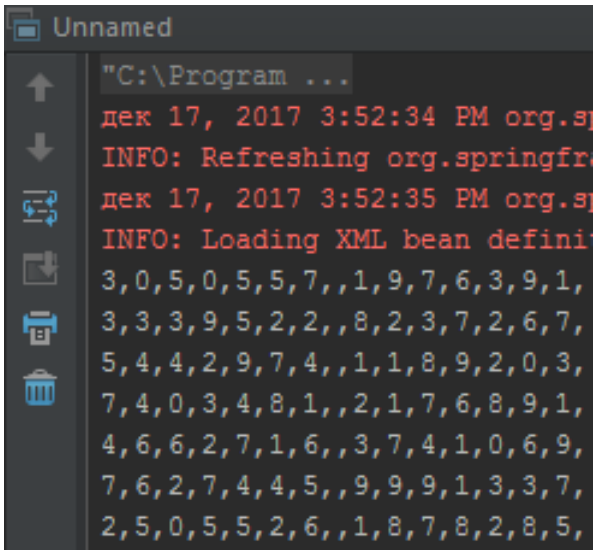


Рисунок 5.6 – Результат выполнения *delimitedIntegerStringGenerator*

Результат выполнения кода со значением бина id="alphanumericStringGenerator" представлен на рисунке 5.7.

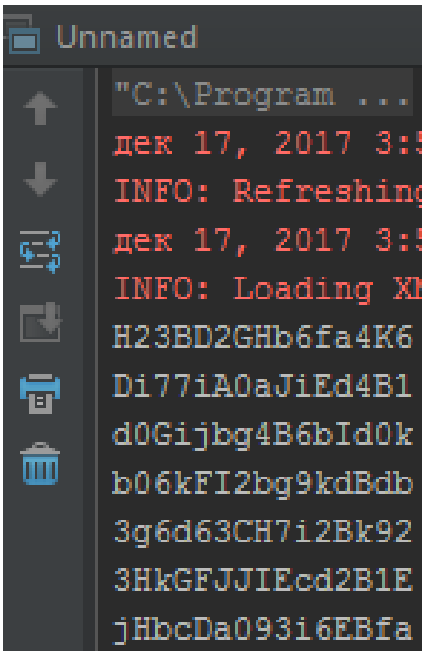


Рисунок 5.7 – Результат выполнения *alphanumericStringGenerator*

5.3 Интеграция *Spring Boot*

Структура проекта (рисунок 5.8).

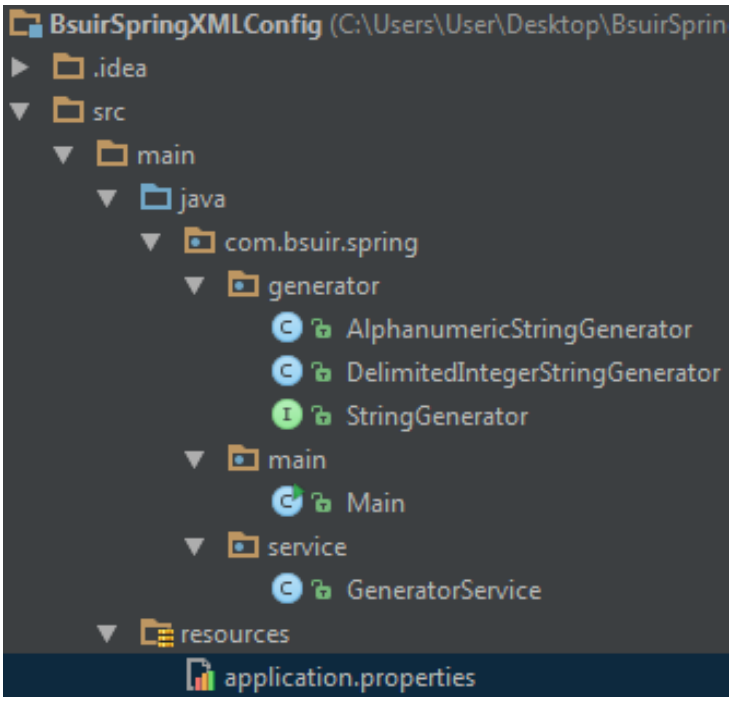


Рисунок 5.8 – Структура проекта

По аналогии создадим *pom.xml*.

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.bsuir.spring</groupId>
<artifactId>BsuirSpringAnnotations</artifactId>
<version>1.0-SNAPSHOT</version>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
                        <addClasspath>true</addClasspath>
                        <mainClass>com.bsuir.spring.main.Main</mainClass>
                        <classpathPrefix>dependency-jars/</classpathPrefix>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                   <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>
                                ${project.build.directory}/dependency-jars/
                            </outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    <properties>
        <spring.version>4.3.9.RELEASE</spring.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

Свойства, необходимые для работы программы, объявлены в файле *application.properties*. Их применение рассмотрим далее.

```
generator.integer.endInclusive:30
generator.integer.delimiter:,
generator.alphanumeric.source:abcdefghijkABCDEFGHIJK123467890
generator.service.strings.count:1
generator.service.string.length:5
```

Пример реализации интерфейса:

С генерацией строки текста:

```
package com.bsuir.spring.generator;
@Component("alphanumericStringGenerator")
public class AlphanumericStringGenerator implements StringGenerator {
    private static final Random RANDOM = new Random();
    @Value("${generator.alphanumeric.source:abcABC123}")
    private String source
    @Override
    public String generate(int count) {
        if (source == null || source.isEmpty()) {
            return EMPTY_STRING;
        }
        if (count < 0) {
            throw new IllegalArgumentException("Length of generated string must
be positive.");
        }
        StringBuilder result = new StringBuilder();
        for (int index = 0; index < count; index++) {
            int sourceIndex = RANDOM.nextInt(source.length());
            result.append(source.charAt(sourceIndex));    }
        return result.toString();
    }}
```

С генерацией строки цифр:

```
package com.bsuir.spring.generator;
@Service("delimitedIntegerStringGenerator")
public class DelimitedIntegerStringGenerator implements StringGenerator {
    private static final Random RANDOM = new Random();
    @Value("${generator.integer.delimiter:,}")
    private String delimiter;
    @Value("${generator.integer.endInclusive:999}")
    private int endInclusive;
    @Override
    public String generate(int count) {
        if (endInclusive <= 0) {
            throw new IllegalArgumentException("Property endInclusive must be
positive integer.");  }
        if (count < 0) {
            throw new IllegalArgumentException("Length of generated string must
be positive.");
        }
        StringBuilder result = new StringBuilder();
        for (int index = 0; index < count; index++) {
            result.append(RANDOM.nextInt(endInclusive)).append(delimiter);
        }
        return result.deleteCharAt(count - 1).toString(); }
}
```

Для создания бинов с помощью аннотаций необходимо добавить к классу одну из следующих аннотаций: *@Component, @Repository, @Service* или *@Controller*. При этом аннотация *@Component* является самой общей, а аннотации *@Repository, @Service* и *@Controller* различаются семантическим значением. Они используются для уровня персистенции, сервисного уровня и уровня представления соответственно. Но, несмотря на различия, они имеют одинаковое действие: сообщают контексту, что необходимо создать бин соответствующего класса. Для рассматриваемого класса будем использовать аннотацию *@Service*. Для связывания свойств используется аннотация *@Value*. В ней можно указать непосредственно значение, которое необходимо поставить, либо, используя специальный синтаксис, можно подставить значение из файла **.property*.

Создадим класс-сервис, осуществляющий прибавление строки к тексту.

```
package com.bsuir.spring.service;
import com.bsuir.spring.generator.StringGenerator;
import org.springframework.stereotype.Service;

@Service
public class GeneratorService {
    @Autowired
    @Qualifier("alphanumericStringGenerator")
    private StringGenerator stringGenerator;
    @Value("${generator.service.strings.count}")
    private int stringsCount;
    @Value("${generator.service.string.length}")
    private int length;
    public List<String> generateStrings() {
        List<String> strings = new ArrayList<>();
        for (int count = 0; count < stringsCount; count++) {
            strings.add(stringGenerator.generate(length));
        }
        return strings;
    }
}
```

Внедрение ссылки на бин осуществляется с помощью аннотации *@Autowired*. Данная аннотация осуществляет связывание по типу, т. е. будет внедрен бин, реализующий интерфейс *StringGenerator*. Если же в контексте зарегистрировано несколько бинов, подходящих под указанный тип, будет сгенерировано исключение *NoUniqueBeanDefinitionException*. Чтобы этого избежать, используется аннотация *@Qualifier*, в которой указывается *id* бина, который необходимо подставить в поле. В данном случае будет подставлен бин

с *alphanumericStringGenerator*. Заметим, что для бинов, созданных с помощью аннотаций, *id* генерируется автоматически как имя класса, записанное с маленькой буквы.

Создадим класс *Main* с точкой входа.

```
package com.bsuir.spring.main;
import com.bsuir.spring.service.GeneratorService;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;
import java.util.List;

@Component
@PropertySource("application.properties")
public class Main {
    private static final String BASE_PACKAGE = "com.bsuir.spring";
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(BASE_PACKAGE);
        GeneratorService service = context.getBean(GeneratorService.class);
        List<String> generated = service.generateStrings();
        for (String s : generated) {
            System.out.println(s);
        }
    }
}
```

В данном классе содержится точка входа (метод *Main*). В данном методе конфигурируется контекст, при этом для всех классов, находящихся внутри пакета на любом уровне вложенности и отмеченных аннотациями или *@Component, @Repository, @Service* и *@Component*, будут созданы бины.

Результат работы зависит от указания *id* бина (*alphanumericStringGenerator* или *delimitedIntegerStringGenerator*) в аннотации *@Qualifier*.

Также отметим, что все свойства, используемые для подстановки с помощью аннотации *@Value*, задаются в файле *application.properties*, который подключается здесь же с использованием аннотации *@PropertySource*. Для того чтобы свойства загрузились корректно, экземпляр данного класса Main также должен быть бином и регистрироваться в контексте, что и делается с помощью аннотации *@Component*.

Список использованных источников

1 Гонсалвес, Э. Изучаем Java EE 7 / Э. Гонсалвес. – СПб. : Питер, 2014. – 640 с.

2 Шилдт, Г. Java. Полное руководство / Г. Шилдт. – М. : Вильямс, 2012. – 1104 с.

3 Хелм, Р. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Р. Хелм, Э. Гамма. – СПб. : Питер, 2013. – 368 с.

4 Хорстманн, К. С. Java. Библиотека профессионала : в 2 т. / К. С. Хортсманн. – 10-е изд. – М. : Вильямс, 2016. – Т. 2 : Расширенные средства программирования. – 864 с.

5 Уоллс, К. Spring в действии / К. Уоллс. – 2-е изд. – М. : ДМК Пресс, 2013. – 752 с.

6 Раджпут, Д. Spring. Все паттерны проектирования / Д. Раджпут. – СПб. : Питер, 2019. – 320 с.

7 Шефер, К. Spring 4 для профессионалов / К. Шефер, Х. О. Кларенс, Р. Харроп. – 4-е изд. – М. : Вильямс, 2015. –752 с.

8 Машнин, Т. С. Web-сервисы Java / Т. С. Машнин [Электронный ресурс]. – СПб. : БХВ-Петербург, 2012. – 560 с. – Режим доступа: https://bookland.com/download/1/10/108012/sample.pdf. – Дата доступа: 21.06.2022.

9 Блинов, И. Н. Java. Методы программирования : учеб.-метод. пособие / И. Н. Блинов, В. С. Романчик. – Минск : Четыре четверти, 2013. – 896 с.

10 Заяц, А. М. Проектирование и разработка WEB-приложений. Введение в frontend и backend разработку на JavaScript и node.js : учеб. пособие / А. М. Заяц, Н. П. Васильев. – СПб. : Лань, 2019. – 120 с.

11 Курняван, Б. Создание web-приложений на языке Java с помощью сервлетов, JSP и EJB / Б. Курняван. – М. : Лори, 2012. – 880 с.

12 Козмина, Ю. Spring 5 для профессионалов / Ю. Козмина, Р. Хаппорт, К. Шефер, К. Хо ; пер. с англ. – СПб. : Диалектика, 2019. – 1120 с.

13 Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. – СПб. : Питер, 2003. – 877 с.

14 Миковски, М. С. Разработка одностраничных веб-приложений / М. С. Миковски, Д. К. Пауэлл. – М. : ДМК Пресс, 2014. – 512 с.

15 Эспозито, Д. Разработка современных веб-приложений: анализ предметных областей и технологий / Д. Эспозито. – М. : Вильямс, 2017. – 464 с.

Св. план 2024, поз. 18

Учебное издание

Петрович Никита Олегович
Голда Ольга
Сторожев Дмитрий Алексеевич

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ РАЗРАБОТКИ СЕРВЕРНОЙ ЧАСТИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *Л. И. Артёмова*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 14.01.2026. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 10,35. Уч.-изд. л. 11,0. Тираж 100 экз. Заказ 63.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск