

МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

Web-программирование

Учебно-методическое пособие
к лабораторным работам

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2022

УДК 681.3.06

Web-программирование: учебно-методическое пособие к лабораторным работам / сост.: М.Н. Гречухин, М.Г. Павловский, Г.В. Разумовский; под общ. ред. Г. В. Разумовского. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2022. 87 с.

Содержат описания лабораторных работ по дисциплине «Web-программирование» по следующим темам: установка и настройка среды разработки и исполнения Web-приложений в среде Eclipse, построение Web-приложений с использованием технологий сервлетов, JSP и GWT, интернационализация приложений, аутентификация и авторизация пользователей, передача данных через файл Cookie и сессионный объект, модульное тестирование Web-приложений.

Предназначены для студентов бакалавриата по направлению № 09.03.01 – «Информатика и вычислительная техника».

Утверждено
редакционно-издательским советом университета
в качестве методических указаний

© СПбГЭТУ «ЛЭТИ», 2022

Лабораторная работа № 1. УСТАНОВКА И НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ И ИСПОЛНЕНИЯ Web-ПРИЛОЖЕНИЯ

Цель работы: изучение процесса настройки Web-сервера Apache Tomcat и системы сборки Maven в среде Eclipse.

Понятие Web-приложения

Web-приложение – это клиент-серверное приложение, в котором в качестве клиента используется браузер, а в качестве сервера – Web-сервер. Клиентская часть реализует пользовательский интерфейс, формирует запросы к серверу и обрабатывает ответы от него. Серверная часть получает запрос от клиента, обрабатывает его, формирует Web-страницу и отправляет ее клиенту по сети с использованием протокола HTTP. Для создания Web-приложений на стороне сервера используются разнообразные технологии и различные языки программирования, но наибольшей популярностью пользуется язык Java, который предлагает широкий спектр решений. Одним из таких решений является Web-сервер Apache Tomcat, в котором реализованы Java-интерфейсы для создания Web-приложений и который легко интегрируется со средой разработки Eclipse.

Загрузка и установка Web-сервера Apache Tomcat

Для загрузки Web-сервера необходимо перейти по ссылке «Download» на страницу <http://tomcat.apache.org/>, на открывшейся странице выбрать последнюю версию Apache Tomcat и из раздела Binary Distributions скачать и распаковать zip-архив. Следует обратить внимание, что для работы Apache Tomcat версии 9.0 и выше требуется установка на компьютере JDK версии не ниже 1.8. Для организации доступа к JDK со стороны Web-сервера Apache Tomcat необходимо добавить системную переменную окружения. Для Windows 10 добавление переменной окружения осуществляется через компонент «Свойства системы» на панели «Этот компьютер». Войдя в этот компонент, необходимо перейти на вкладку «Дополнительные параметры системы» и нажать кнопку «Переменные среды», после чего откроется окно, изображенное на рис. 1.1.

Если в списке системных переменных не прописан доступ к JDK, необходимо добавить новую переменную окружения, для чего на панели «Переменные среды пользователя для User» нажать кнопку «Создать» и в открывшемся окне ввести имя переменной (JAVA_HOME). Для ввода расположе-

ния JDK, надо выделить на панели «Системные переменные» строку «Path» и нажать кнопку «Изменить». В появившемся окне (рис. 1.2) ввести путь к каталогу bin JDK.

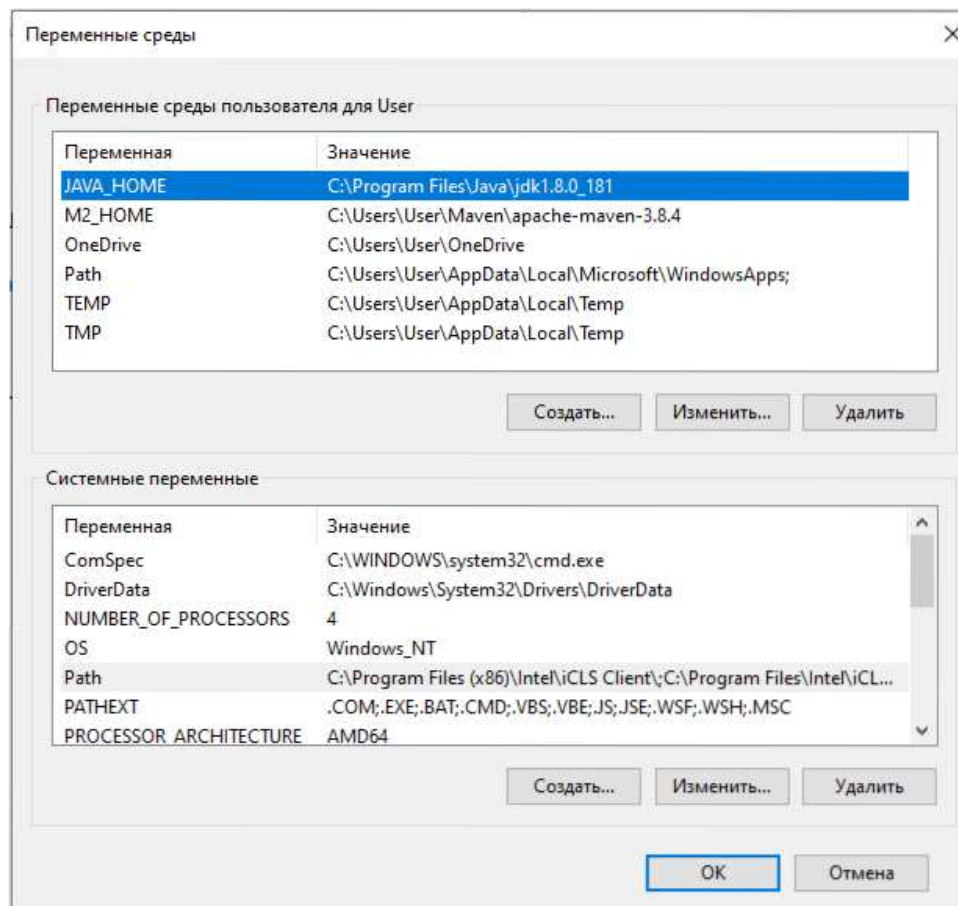


Рис. 1.1. Окно редактирования переменных окружения

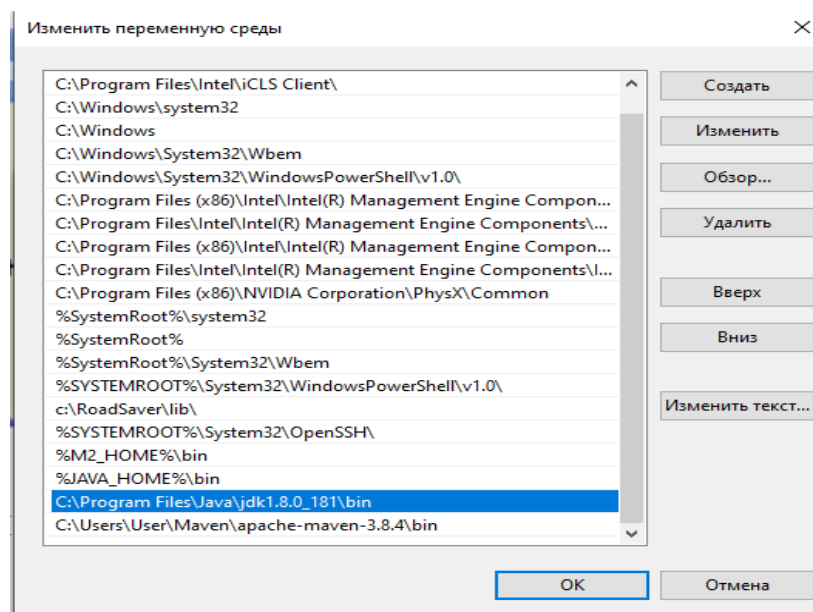


Рис. 1.2. Задание значений системной переменной окружения

В распакованном архиве сервера Tomcat в каталоге bin располагаются файлы запуска и остановки Web-сервера. Запуск Web-сервера осуществляется исполнением файла startup для Windows и startup.sh – для Linux, останов Web-сервера – соответственно исполнением файла shutdown и shutdown.sh.

Для проверки работоспособности Web-сервера после его запуска в браузере, например Яндекс, вводим адрес <http://localhost:8080>. Если сервер установлен правильно, то появится следующая HTML-страница (рис. 1.3):

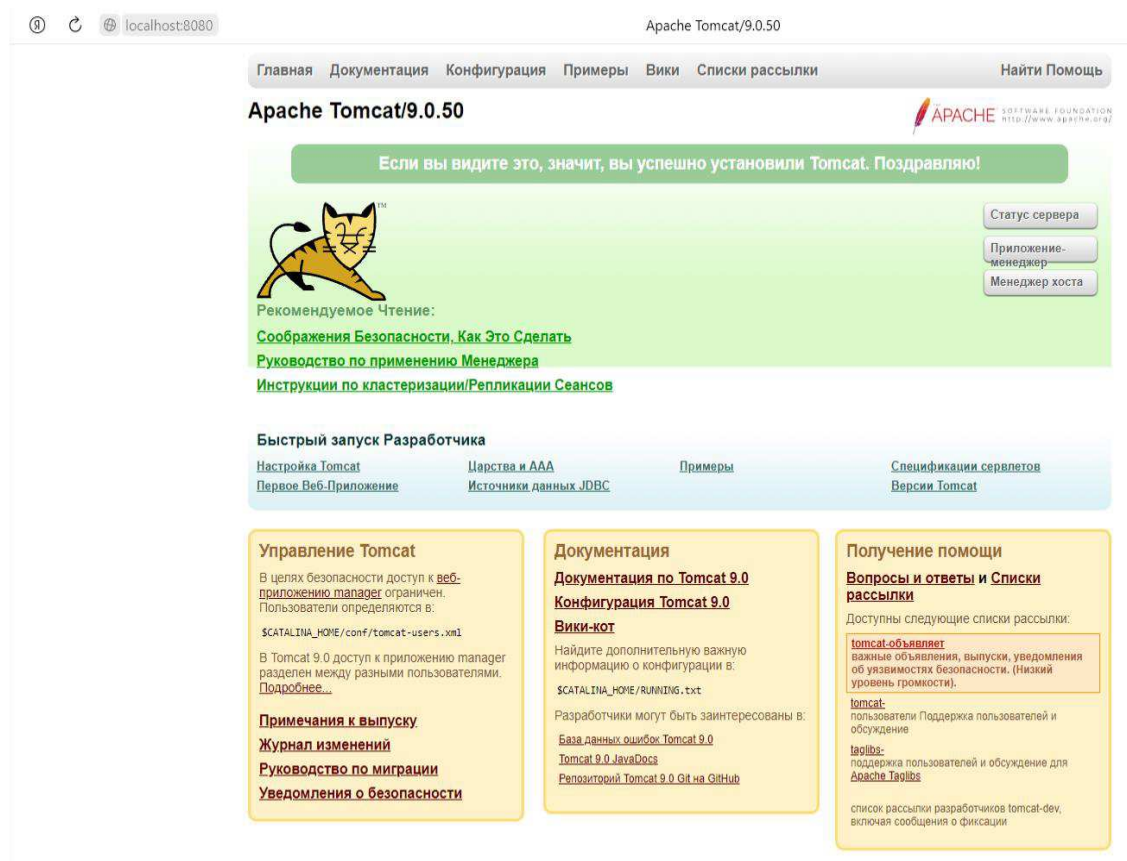


Рис. 1.3. HTML-страница при нормальном запуске Web-сервера

Если сервер не работает, то появится сообщение (рис.1.4):

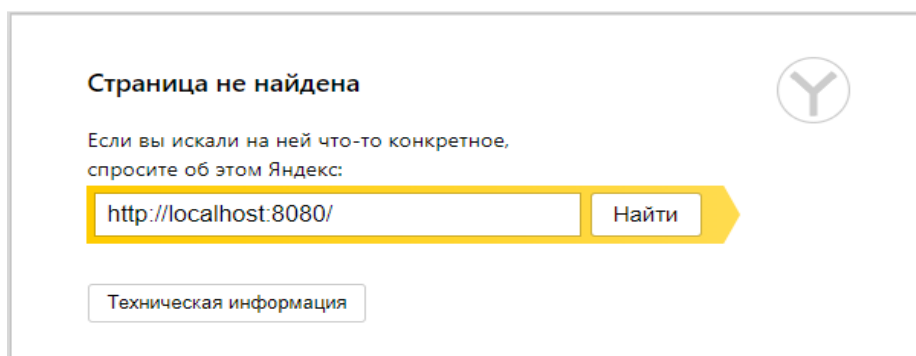


Рис. 1.4. Сообщение при неудачном запуске Web-сервера

По умолчанию Apache Tomcat использует для приема входящих подключений TCP-порт 8080, а также порты 8009 и 8005. Чтобы Web-сервер принимал сообщения по другому порту, например порту 80, необходимо отредактировать файл конфигурации `\conf\server.xml`. Кроме того, рекомендуется добавить в файл конфигурации параметр `useBodyEncodingForURI="true"`, который позволит использовать в параметрах запроса русские буквы. Чтобы внести эти изменения, исходную строку `<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />` надо заменить на строку `<Connector port="80" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" useBodyEncodingForURI="true"/>`. После перезапуска Web-сервера его доступность по порту 80 можно проверить, обратившись по адресу <http://localhost/>. Если порт занят другим приложением, то будет выведено сообщение (рис. 1.5):

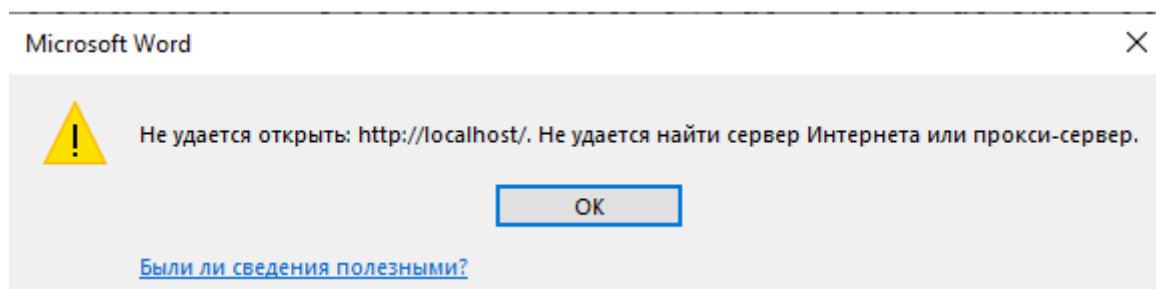


Рис. 1.5. Сообщение при занятости порта

Узнать, какое приложение использует требуемый порт, можно с помощью команды: **netstat -na**.

Найденное приложение надо остановить и перезапустить Web-сервер, после чего снова обратиться к нему по требуемому адресу. При правильной работе Web-сервера он отобразит такую же HTML-страницу, как на рис. 1.3.

Настройка Eclipse на работу с Web-сервером Apache Tomcat

Для разработки Web-приложений в среде Eclipse необходимо подключить к ней Web-сервер. Это позволит использовать в проекте классы Web-сервера Apache Tomcat и непосредственно из среды разработки управлять его настройками и запуском.

Для подключения Web-сервера надо запустить Eclipse, выбрать пункт меню Window → Preferences → Server → Runtime Environments и нажать кнопку «Add». В открывшемся окне выбрать версию Apache Tomcat (в данном случае это Apache Tomcat v9.0) и нажать кнопку «Next» (рис. 1.6). В заключительной части настройки (рис. 1.7) надо указывают путь к каталогу Apache Tomcat и среду выполнения Java, в которой должен выполняться Web-сервер (JDK 1.8 или выше). После этого надо нажать кнопку «Finish».

Для включения Web-сервера в среду исполнения на панели меню Eclipse надо перейти в пункт меню File→ New →Other и на панели *Select a wizard* (рис. 1.8) выбрать Server -> Server, а затем нажать кнопку «Next». В появившемся окне (рис. 1.9) задать тип сервера и нажать кнопку «Finish».

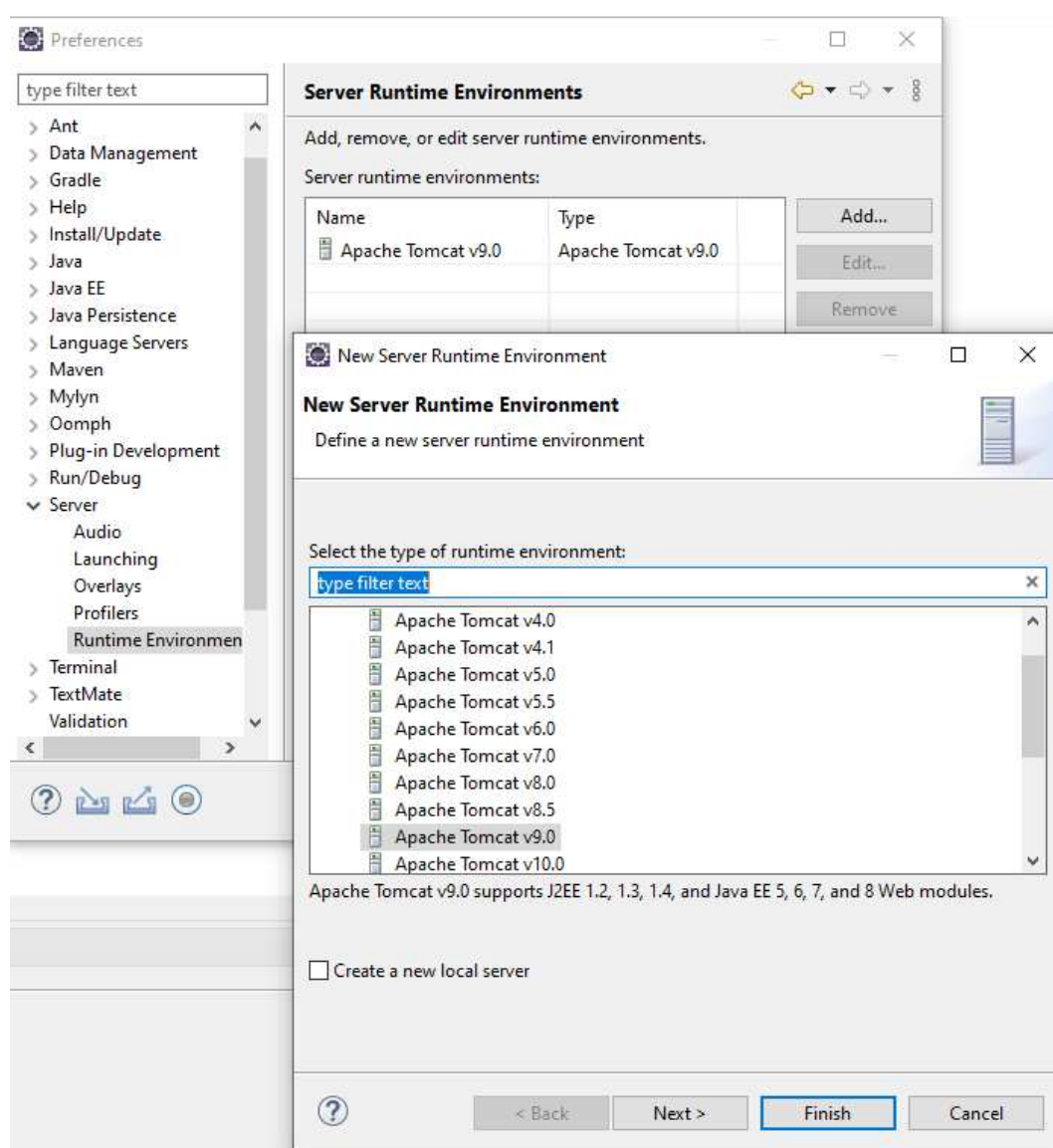


Рис. 1.6. Выбор версии Apache Tomcat

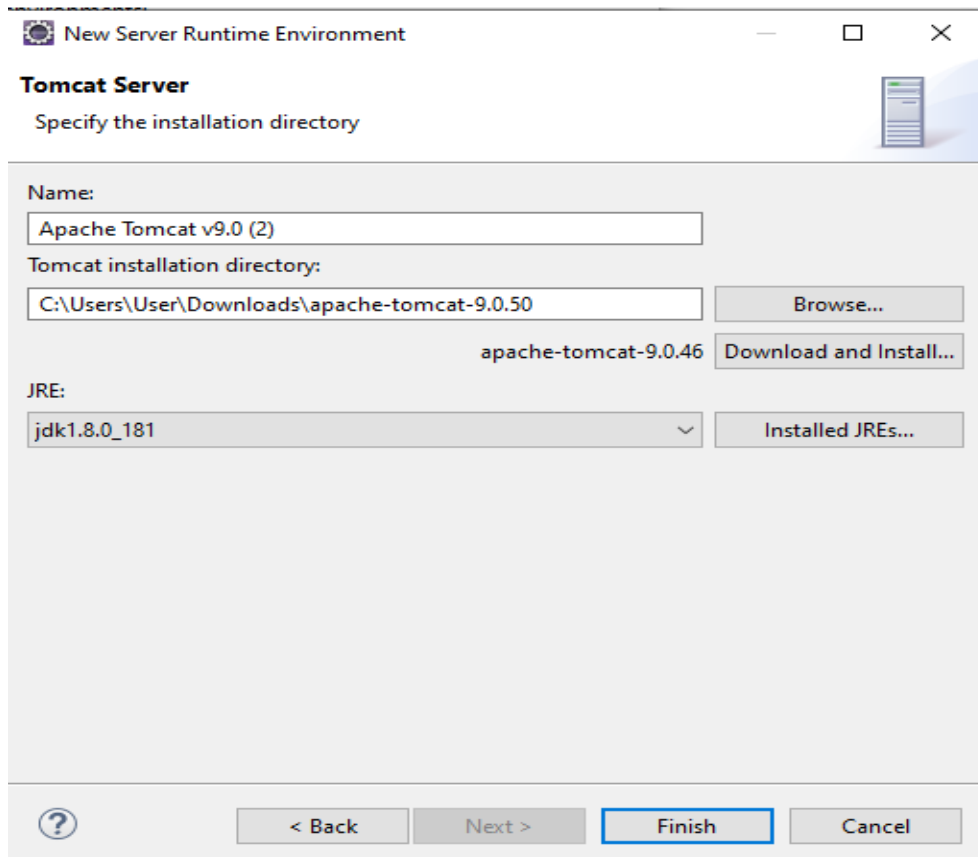


Рис. 1.7. Задание пути к каталогу Apache Tomcat и среды выполнения Java

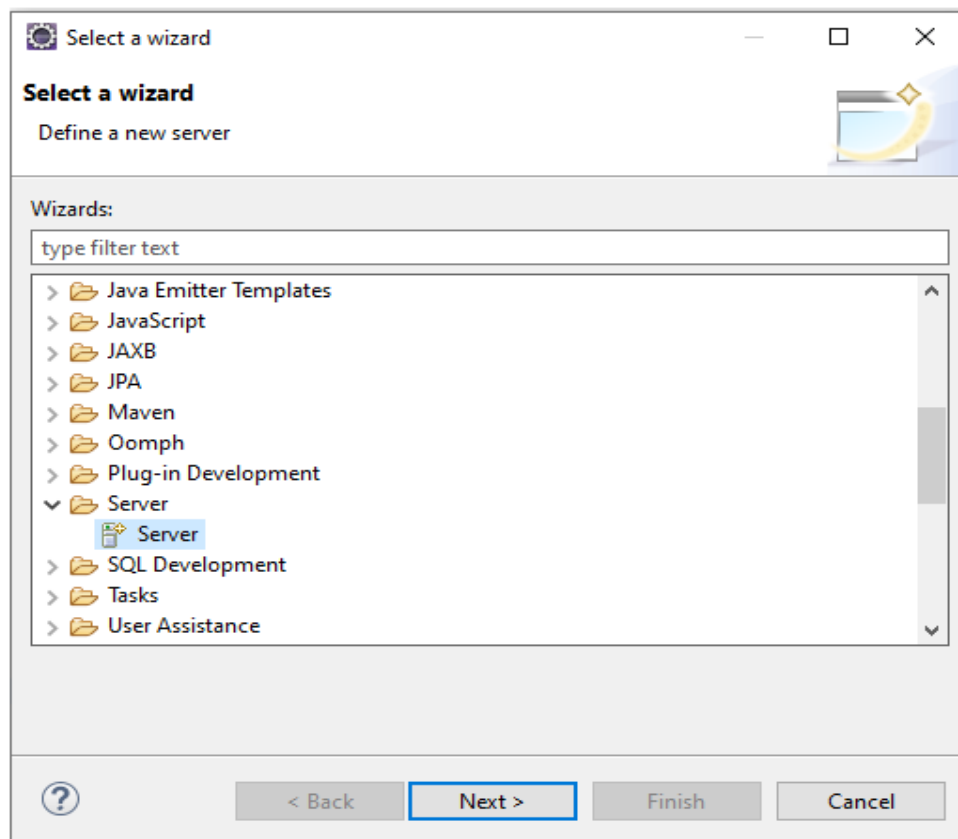


Рис. 1.8. Добавление нового сервера

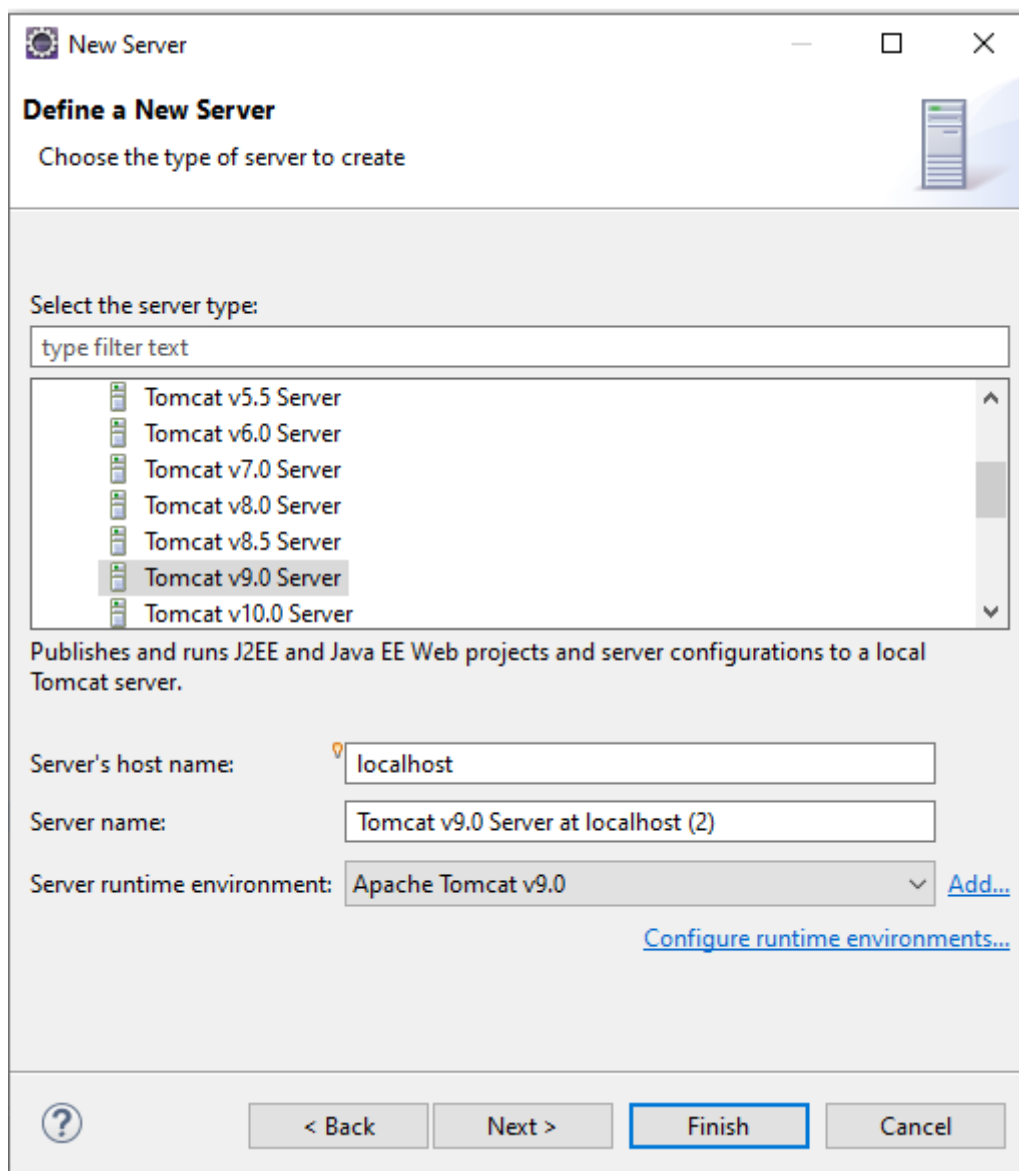


Рис. 1.9. Выбор типа Web-сервера

После проделанных действий на панели **Package Explorer** должен появиться каталог **Servers**, а в нем подкаталог (Tomcat v9.0 Server at localhost-config) созданного профиля со всеми файлами настроек, необходимыми для конфигурации Apache Tomcat:

- server.xml (общие настройки сервера);
- web.xml (параметры, общие для всех Web-приложений; настройки отдельных Web-приложений задаются в их собственных файлах /WEB-INF/Web.xml);
- context.xml (общие настройки управления контентом);
- tomcat-users.xml (управление пользователями и ролями).

Для настройки сервера можно открыть в редакторе любой из этих файлов и сконфигурировать Apache Tomcat, как это необходимо. Также на панели **Console** появится закладка **Servers**, в которой будет прописана строка Tomcat v9.0 Server at localhost [Stopped]. После установки курсора на этой строке и щелчка правой клавишей мыши появится меню, с помощью которого этот сервер можно запускать, останавливать, перезапускать, запускать в режиме отладки и т. п. Если на строке 2 раза щелкнуть левой клавишей мыши, то откроется окно, через которое можно установить некоторые настройки работы Tomcat, например порты, каталог для развертывания, тайм-аут и др. (рис. 1.10).

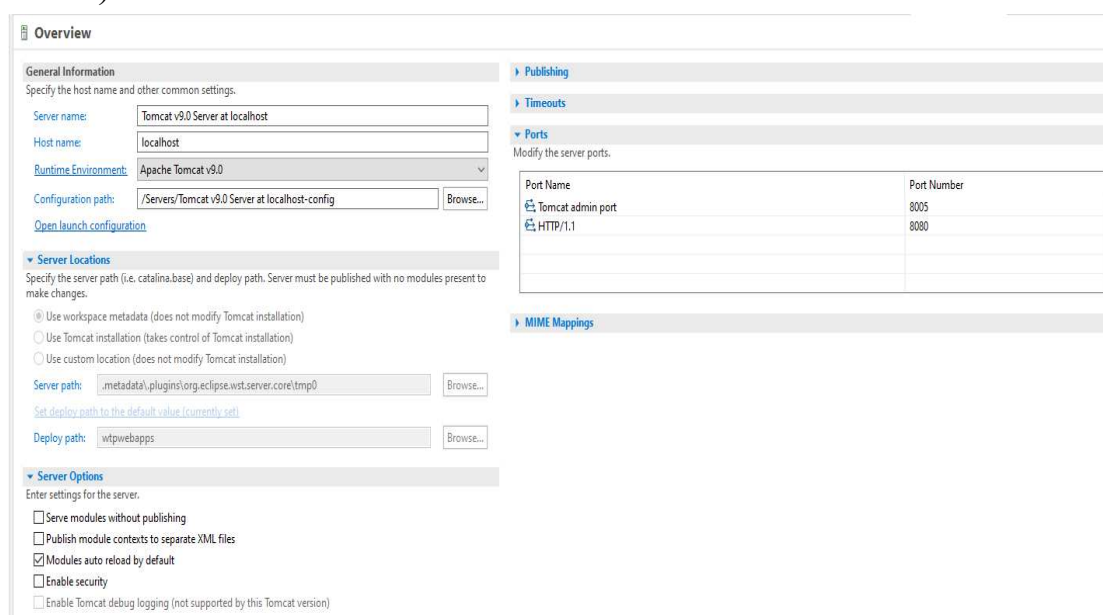


Рис. 1.10. Настройки Web-сервера

Подключаемый модуль Eclipse следит за состоянием Web-сервера и отображает индикаторы stopped, starting и started в области вывода данных программы.

Загрузка и установка системы сборки Maven

Система сборки Maven это инструмент автоматизации сборки проектов. На вход система сборки получает исходный код, библиотеки и ресурсы, а на выход выдаёт программу, которую можно запустить. Для использования системы сборки необходимо создать maven-проект. В Eclipse есть две возможности создавать и импортировать maven-проекты в среду разработки. Это использовать встроенный (EMBEDDED) плагин или внешний, отдельно установленный фреймворк Maven.

Посмотреть какой сборщик доступен в Eclipse можно воспользовавшись меню Window\Preferences\Maven\Installations (рис.1.11)

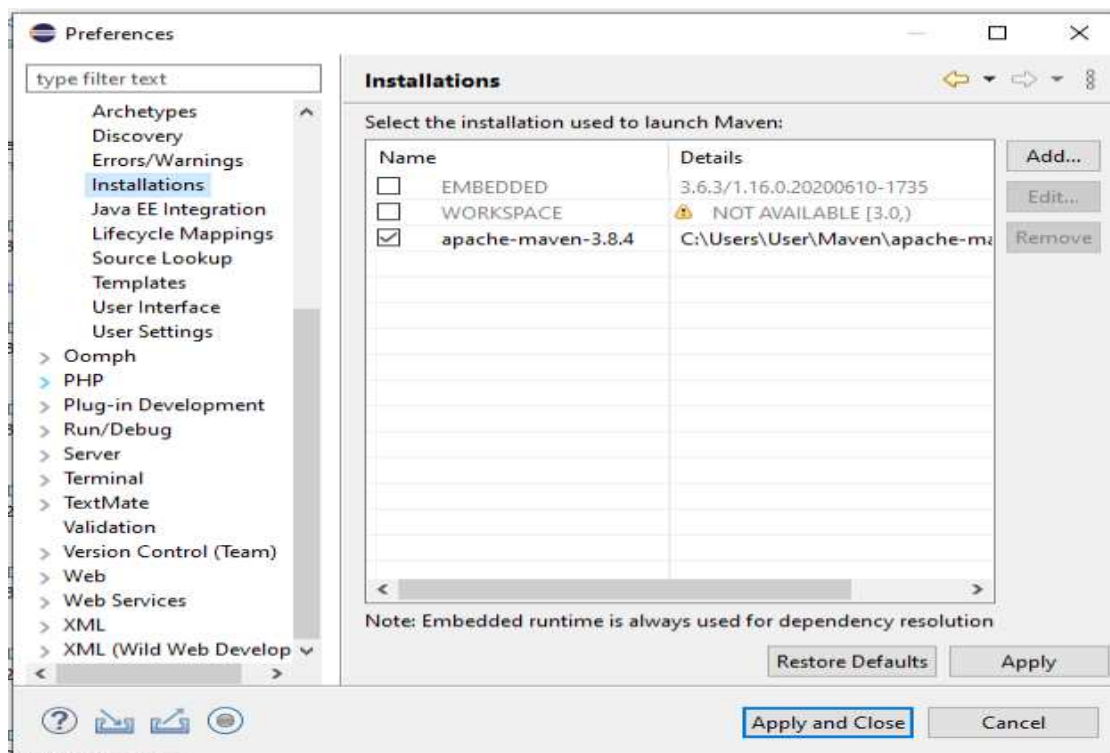


Рис. 1.11. Окно инсталляции Maven

Для установки последней версии фреймворка надо скачать двоичный zip-архив с сайта <https://maven.apache.org/> и распаковать архив в любую директорию. Далее необходимо в системной переменной Path указать путь к каталогу Maven\bin. Проверить правильность установки Maven можно выполнив в терминале следующую команду:

```
C:\Users\User>mvn -v
```

Должна появиться информация о версии Maven, jre и операционной системе (рис. 1.12).

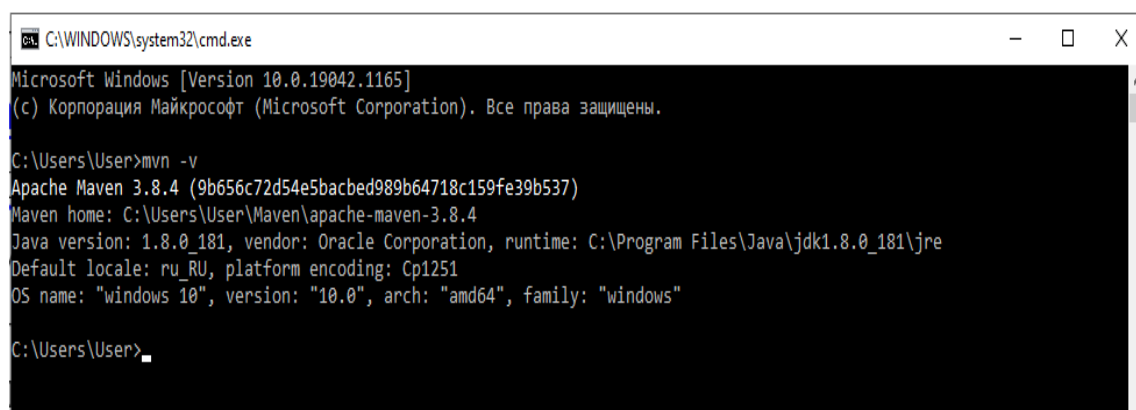


Рис. 1.12. Информация установки Maven

После установки Maven его можно запускать либо через командную строку, либо в среде разработки. Для добавления в Eclipse установленного Maven необходимо в окне Installations воспользоваться кнопкой Add. Появится окно New Maven Runtime (рис. 1.13) и в строке Installation home надо указать путь к каталогу Maven. Добавленный внешний Maven должен быть отмечен в окне Installations (рис.1.11) галочкой в качестве основного инструмента для сборки.

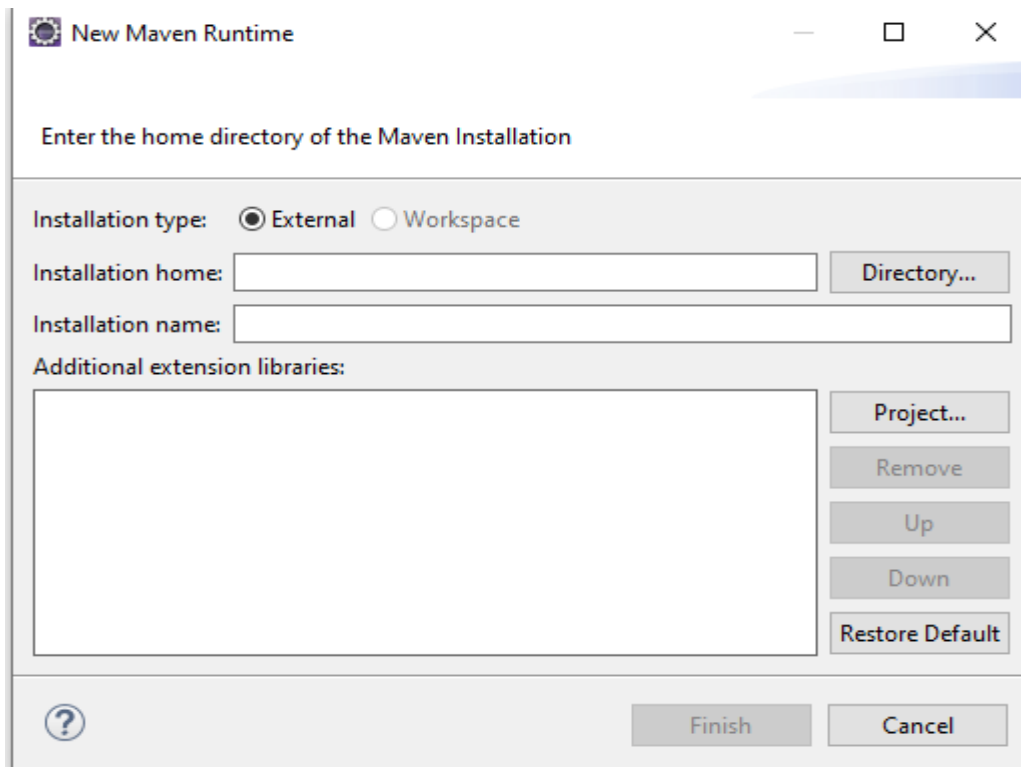


Рис. 1.13 Окно подключения Maven

Порядок выполнения лабораторной работы

1. Загрузите и установите на свой компьютер Web-сервер Apache Tomcat.
2. Внесите изменения в файл server.xml.
3. Проверьте работоспособность Web-сервера по порту 8080 и какому-нибудь другому порту.
4. Настройте Eclipse на работу с Web-сервером Apache Tomcat.
5. Запустите и остановите Web-сервер Apache Tomcat из среды Eclipse.
6. Загрузите и установите в Eclipse фреймворк Maven.
7. Проверьте правильность установки Maven с помощью команды `mvn-v`.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Перечень основных действий и скриншоты, подтверждающие установку Web-сервер Apache Tomcat и фреймворка Maven в среду Eclipse.
2. Распечатку измененного файла server.xml.
3. Скриншот установленной версии Maven.

Лабораторная работа № 2. СОЗДАНИЕ Web-ПРОЕКТА С ИСПОЛЬЗОВАНИЕМ СИСТЕМЫ СБОРКИ MAVEN

Цель работы: знакомство с основными понятиями сборки и средствами создания web-проекта с помощью Maven.

Создание web-проекта с помощью Maven

Для создания Maven проекта в Eclipse необходимо выбрать пункты меню File→NEW→Project и в появившемся окне раскрыть секцию Maven, в которой выделить пункт Maven Project. Затем нажать кнопку Next в текущем и следующем окне. Появится окно выбора архетипа (рис.2.1).

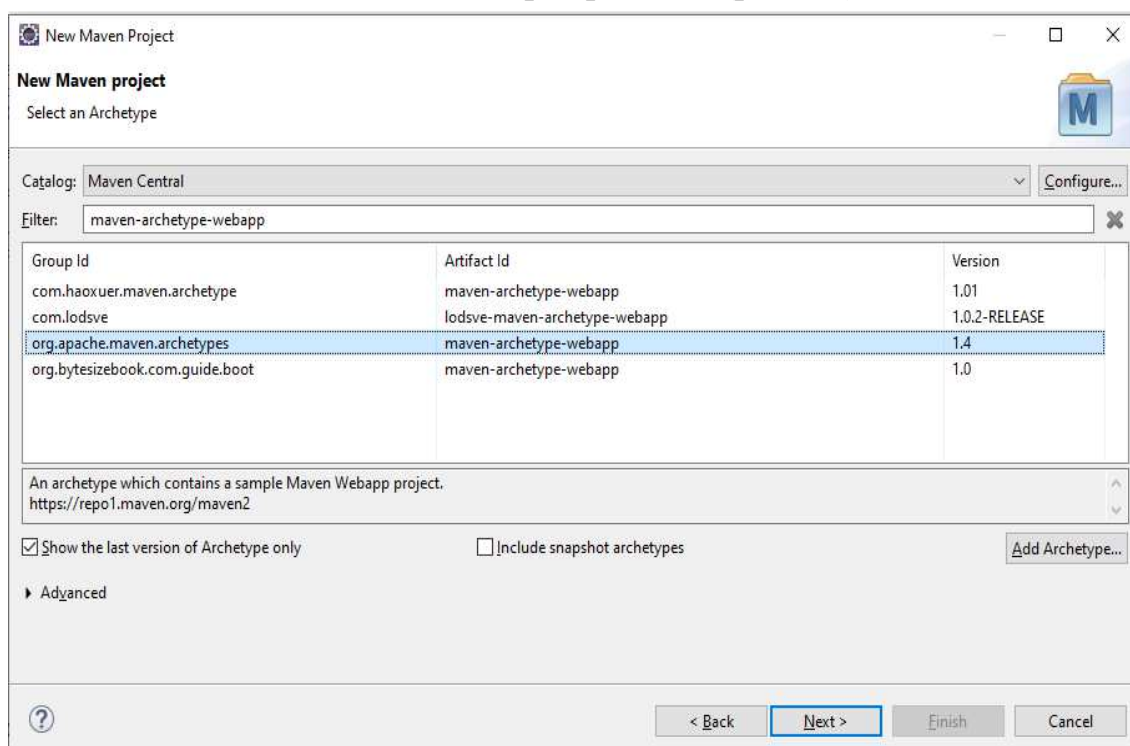


Рис. 2.1 Окно выбора архетипа

Архетип это шаблон, который задает начальную структуру проекта. В Eclipse определены более 1000 архетипов. Для создания простого веб-проекта надо выбрать архетип maven-archetype-webapp 1.4 и нажать кнопку

Next. В раскрывшемся окне (рис. 2.2) надо задать параметры: Group Id (данные о компании и авторах проекта, например NIC), Artifact Id (идентификатор проекта, например mvnweb1) и Version (номер версии проекта, по умолчанию 0.0.1-SNAPSHOT). Обозначение SNAPSHOT означает, что проект находится в активной фазе разработке и в него вносятся постоянные изменения. В этом случае Maven каждый раз при сборке будет заново загружать библиотеки, у которых в версии указано SNAPSHOT. После заполнения указанных полей надо нажать кнопку Finish.

New Maven Project

New Maven project

Specify Archetype parameters

Group Id: NIC

Artifact Id: mvnweb

Version: 0.0.1-SNAPSHOT

Package: NIC.mvnweb

Properties available from archetype:

Name	Value

Advanced

< Back Next > Finish Cancel

Рис. 2.2 Окно спецификации параметров проекта

По этому архетипу Maven сгенерирует иерархию стандартных каталогов (рис. 2.3), pom.xml файл и jsp страницу index.jsp, размещенную в папке webapp. Файл index.jsp качестве образца содержит html-код вывода строки Hello world. В проект также включается каталог target, куда после инсталляции проекта будут помещены скомпилированные файлы. Перечень стандартных каталогов, которые может создавать Maven, можно посмотреть на <https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html>.

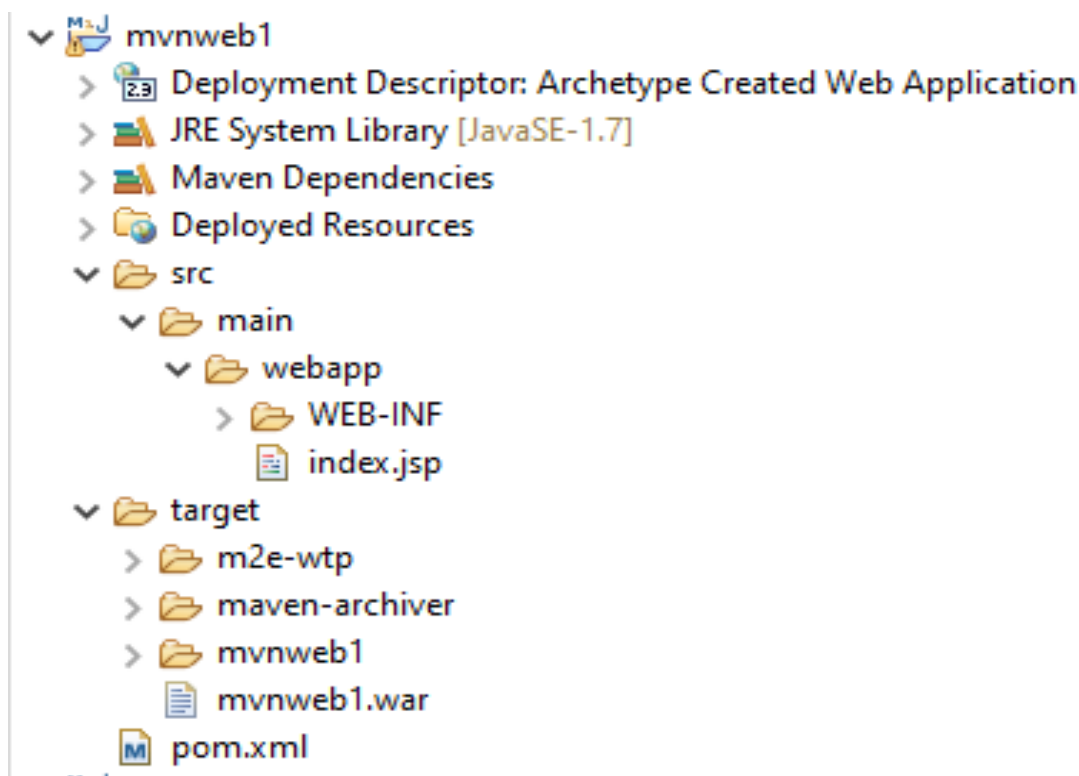


Рис. 2.3 Структура директорий Maven web-проекта

Структура файла pom.xml

В корне проекта помимо стандартных каталогов также включается описание проекта, располагающееся в файле `pom.xml`. Это конфигурационный файл, который отражает объектную модель проекта, содержит настройки процесса сборки, обладает иерархической структурой (есть родительские настройки и переопределяемые наследниках). Родительский супер `pom.xml` содержит все настройки по умолчанию, аналог класса `Object` в Java. Он располагается внутри jar-файла установленного фреймворка `Maven\apache-maven-3.8.4\lib\maven-model-builder-3.8.4.jar\org\apache\maven\model\`. От него могут наследоваться другие `pom.xml` файлы.

Файл `pom.xml` может содержать следующие секции:

- организация, название проекта и его версия (теги `groupId`, `artifactId`, `version`);
- путь к репозиторию проекта, откуда его можно скачать (тег `url`);
- зависимости проекта (тег `dependencies`);
- версии библиотек, плагинов (тег `properties`);
- сборка (тег `build`);
- функциональные модули сборки (тег `plugin`).

Зависимости - это библиотеки кода, которые хранятся в репозиториях и необходимы для выполнения приложения и его тестирования. Репозиторий представляет из себя набор папок, где у каждой зависимости есть адрес хранения. Репозитории делятся на локальные (по умолчанию расположен на компьютере в папке C:\Users\User\.m2\repository) и удаленные (в сети интернет). Добавить дополнительный локальный репозиторий можно в файле settings.xml, расположенного в каталоге C:\Users\User\Maven\apache-maven-3.8.4\conf. Maven скачивает зависимости из удаленных репозиторий в локальный и работает при необходимости с библиотеками, расположенными в локальном репозитории. По умолчанию удаленный репозиторий определен в корневом pom.xml как Central Repository.

```
<repository>
  <id>central</id>
  <name>Central Repository</name>
  <url>https://repo.maven.apache.org/maven2</url>
  <layout>default</layout>
  <snapshots>
    <enabled>>false</enabled>
  </snapshots>
</repository>
```

Если в удаленном репозитории отсутствуют необходимые библиотеки, либо удаленный репозиторий оказывается недоступным, то в корневом pom.xml можно указать другой удаленный репозиторий. У каждой зависимости в локальном репозитории есть свой pom.xml файл, в котором могут быть прописаны другие транзитивные зависимости и координаты проекта. По этим координатам Maven определяет наличие или отсутствие библиотек в репозитории.

При описании зависимости надо указать идентификатор производителя, имя библиотеки, номер версии (теги groupId, artifactId, version). Не обязательными параметрами могут быть область действия библиотеки (тег scope) и классификатор, уточняющий использование библиотеки (тег classifier). По умолчанию область действия compile (зависимость доступна во всем проекте). Например, если зависимость junit имеет область действия test, то эта зависимость будет использована maven при выполнении компиляции той части проекта, которая содержит тесты, а также при запуске тестов на выполнение и построении отчета с результатами тестирования кода. Ниже приведен пример описания зависимости для использования библиотеки Junit.

```
<dependency>
  <groupId>junit</groupId>
```



```

    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>

```

Сборка Maven проекта состоит из нескольких последовательных этапов. Основными этапами (фазами) сборки являются:

Clean - удаление каталогов и файлов из каталога target;

Validate - подтверждает, является ли проект корректным и вся ли необходимая информация доступна для завершения процесса сборки.

Resources - копирование ресурсов в каталог target;

Compile - компиляция файлов;

Test - запуск тестов;

Package - упаковка скомпилированных файлов в архивы;

Install - установка проекта в локальный репозиторий;

Deploy - копирование архива в удаленный репозиторий;

Site - создается документация проекта.

Каждый из этих этапов может иметь фазы **pre** и **post**. Они могут быть использованы для регистрации задач, которые должны быть запущены перед и после указанной фазы.

Сборка определяется в секции `<build></build>` файла `pom.xml`. Ниже приведен пример секции `<build>`, сгенерированной для архетипа `maven-archetype-webapp`.

```

<build>
  <finalName>mvnweb1</finalName>
  <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be
moved to parent pom) -->
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
      <!-- see http://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin\_bindings\_for\_war\_packaging -->
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
    </plugins>
  </pluginManagement>

```

```

    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.1</version>
  </plugin>
  <plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.2.2</version>
  </plugin>
  <plugin>
    <artifactId>maven-install-plugin</artifactId>
    <version>2.5.2</version>
  </plugin>
  <plugin>
    <artifactId>maven-deploy-plugin</artifactId>
    <version>2.8.2</version>
  </plugin>
</plugins>
</pluginManagement>
</build>

```

В этой секции последовательно перечисляются плагины, реализующие этапы сборки. Для сборки могут быть использованы как Maven плагины, так и собственные. Базовый набор плагинов Maven и документация по ним можно найти на сайте <http://maven.apache.org/plugins/index.html>. Список установленных на компьютере плагинов Maven можно увидеть в директории `\.m2\repository\org\apache\maven\plugins`. При объявлении плагина его можно настроить так, чтобы он автоматически запускался в нужный момент и выполнял определенную команду. Это задается соответственно в тегах `<phase>` и `<goal>`. К каждой фазе можно привязать несколько команд, а можно ни одной — тогда фаза просто пропускается при сборке. У плагина также может существовать фаза по умолчанию, заданная при программировании плагина.

Плагин *maven-clean-plugin* используется, когда надо удалить файлы, созданные во время сборки. У плагина только одна команда `clean`.

Плагин копирования ресурсов *maven-resources-plugin* имеет команду `copy-resources`, которая позволяет все ресурсы (файлы изображений, файлы `.properties`) скопировать в директорию *target*.

Плагин *maven-compiler-plugin* имеет две команды: `compile` - компиляция исходников, по умолчанию связана с фазой `compile` и `testCompile` - компиляция тестов, по умолчанию связана с фазой `test`.

Плагин *maven-surefire-plugin* предназначен для запуска тестов и генерации отчетов по результатам их выполнения. Он содержит единственную команду *test*. По умолчанию на тестирование запускаются все `java`-файлы,

наименование которых начинается с «Test» и заканчивается «Test» или «TestCase». Если необходимо запустить java-файл с отличным от соглашения наименованием, то необходимо в проектный файл pom.xml включить соответствующую секцию. Результаты тестирования в виде отчетов в форматах .txt и .xml сохраняются в директории /target/surefire-reports.

Плагин *maven-war-plugin* отвечает за сбор всех зависимостей артефактов, классов и ресурсов веб-приложения и их упаковку в архив веб-приложения. По умолчанию он запускается с командой `war`, которая создает файл `war`. В нем содержатся в формате JAR компоненты web-приложения.

Плагин *maven-install-plugin* используется для добавления артефактов в локальный репозиторий. Основной командой этого плагина является `install`, которая по умолчанию привязана к этапу установки. Другие команды – `install-file`, используемая для автоматической установки внешних артефактов в локальный репозиторий, и `help`, отображающая информацию о самом плагине.

Для выполнения этапа копирования архива в удаленный репозиторий необходима ручная настройка плагина *maven-deploy-plugin*, где должен быть указан путь к удаленному репозиторию (по умолчанию путь к удаленному репозиторию находится в корневом pom.xml). В большинстве сборок проекта фаза развертывания реализуется с помощью команды `deploy`. Кроме того, артефакты, которые не созданы с помощью Maven, могут быть добавлены в удаленное хранилище с помощью `deploy -file`.

Maven по умолчанию настроен для создания отчетов о проекте и размещении их на сайте. Для создания сайта используется плагин *maven-site-plugin*, а контента плагин *maven-project-info-reports-plugin*. Плагин *maven-javadoc-plugin* предназначен для того, чтобы генерировать документацию по исходному коду проекта стандартной утилитой *javadoc*.

Создание сборки web-приложения

Для создания сборки необходимо нажать правую кнопку мыши на имени проекта и в появившемся окне выбрать пункт `Run As` и вариант работы с проектом (рис.2.4). Eclipse предлагает следующие варианты работы с проектом:

1. Run on Server
2. Java Application
3. Unit Test
4. Maven build M

5. Maven build
6. Maven clean
7. Maven generate-sources
8. Maven install
9. Maven test

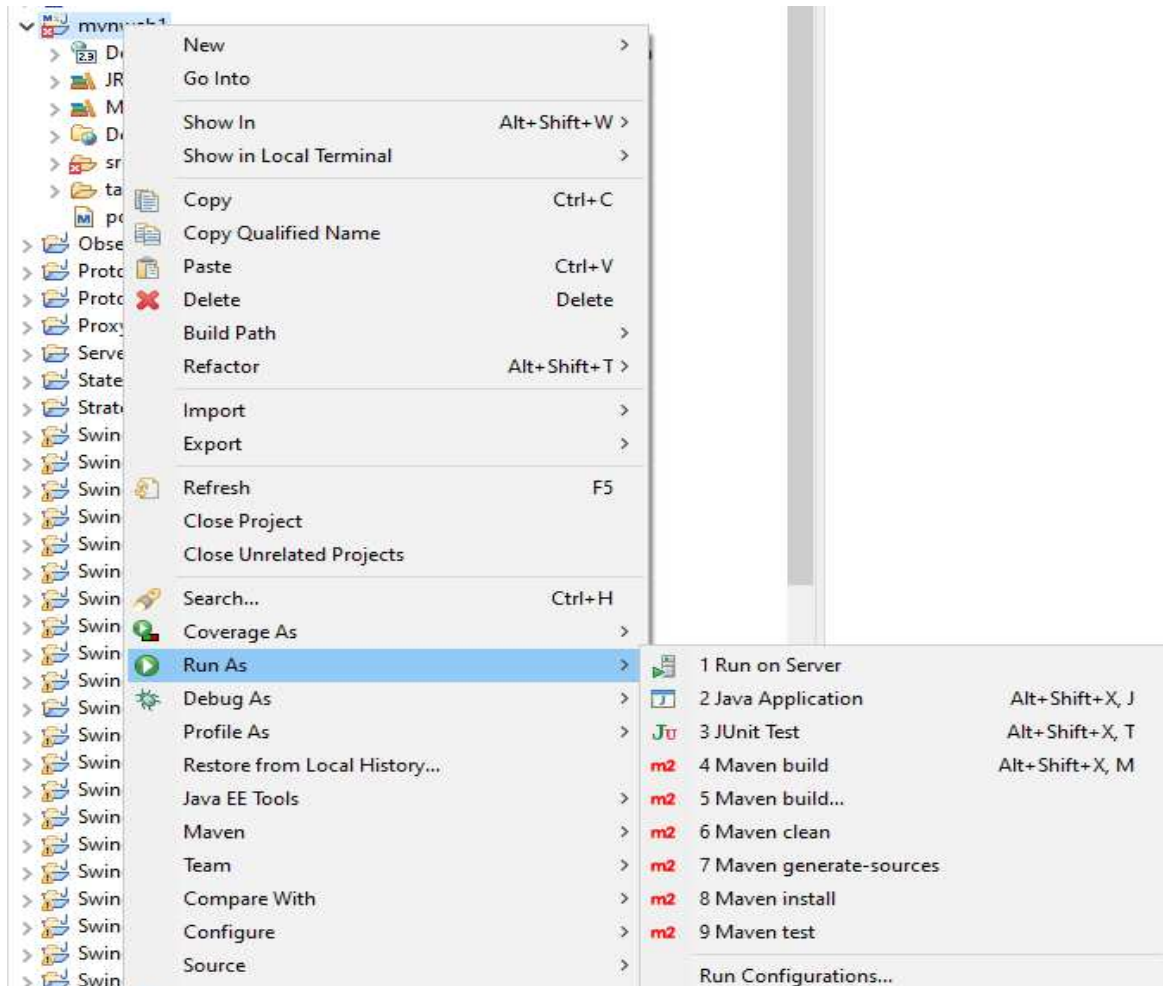


Рис. 2.4 Окно выбора сборки

Результаты выполнения этих команд можно увидеть в консоли. Первая команда запускает сервер Tomcat, который читает из папки WEB-INF файл html или jsp web-приложения. Вторая и третья команды связаны с запуском java-приложения и тестов, находящихся соответственно в каталогах java и test. Оба этих каталогов надо создать в директории main и поместить туда исходники java-приложения и тесты. Следующие шесть команд предлагают различные сборки. В pom.xml файле не указано, что надо собрать war-файл. Его состав определяется командами, которые указываются при запуске сборки.

По команде clean будет запущен плагин maven-clean-plugin:3.1.0:clean (default-clean) и выполнена очистка каталога target.

По команде `test` будут выполнены последовательно этапы копирования ресурсов для приложения, компиляция приложения, копирование ресурсов для тестов, компиляция тестов, запуск теста и соответственно запущены плагины `maven-resources-plugin:3.0.2:resources` (default-resources), `maven-compiler-plugin:3.8.0:compile` (default-compile), `maven-resources-plugin:3.0.2:testResources` (default-testResources), `maven-compiler-plugin:3.8.0:testCompile` (default-testCompile) и `maven-surefire-plugin:2.22.1:test` (default-test).

Так как для команды `generate-sources` плагин в `pom.xml` файле не определен, то по этой команде никаких действий выполняться не будет.

По команде `install` сначала выполняться этапы как у команды `test`, а затем фазы создания архива и размещения его в локальном репозитории. Для этого будут запущены плагины `maven-war-plugin:3.2.2:war` (default-war) и `maven-install-plugin:2.5.2:install` (default-install).

При выборе варианта `Maven build` открывается окно (рис. 2.5), где в поле `Goals` можно задать конфигурацию сборки.

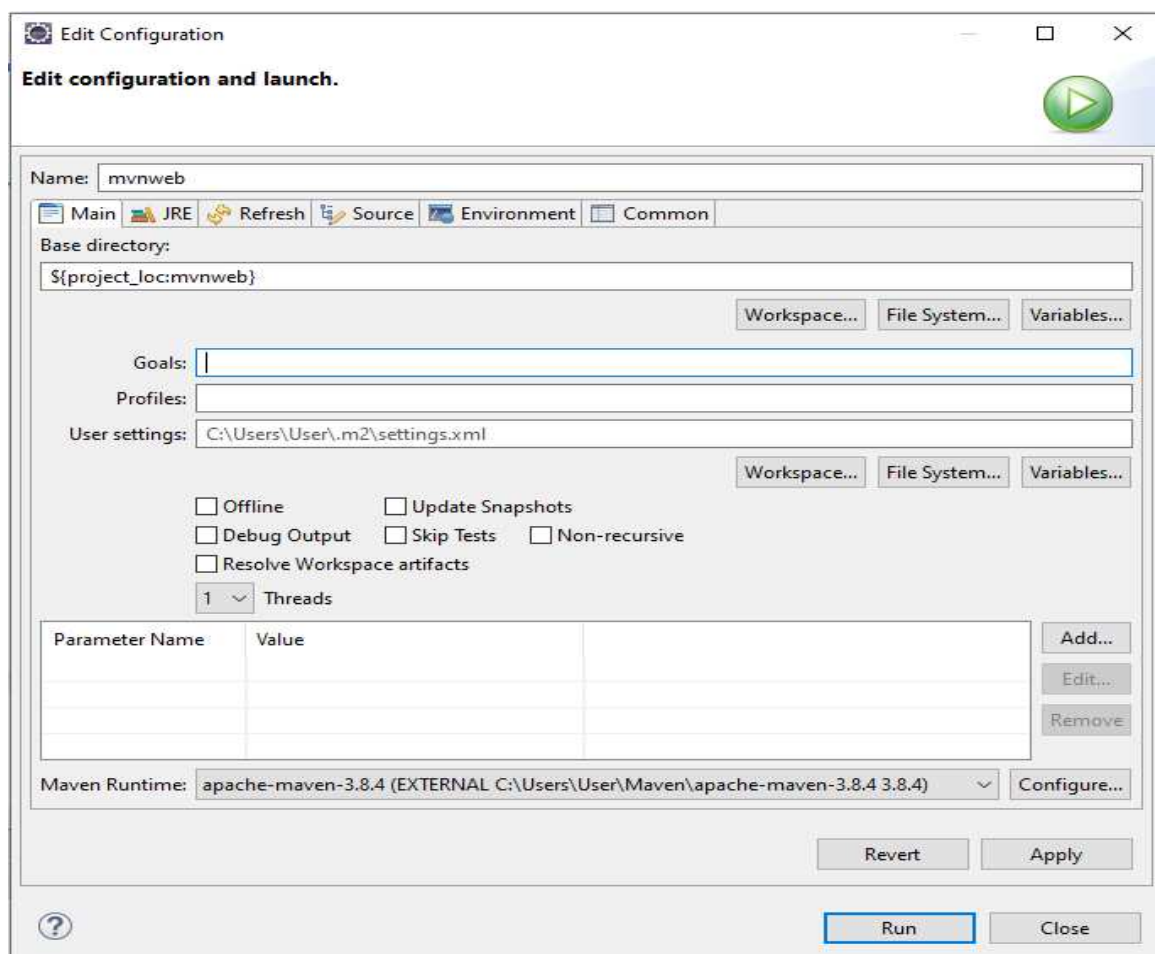


Рис. 2.5 Окно конфигурации сборки

Например, в поле Goals можно перечислить команды, которые надо выполнить при сборке, или поменять расположение настроечного файла settings.xml.

После выполнения команды Maven install в каталоге target появится файл с расширением war. Файл WAR - это веб-приложение, упакованное в формате архива и предназначенное для выполнения на веб-сервере. Этот файл необходимо скопировать в каталог (webapps) веб-сервера Tomcat после чего перезапустить веб-сервер. Копирование выполняется через меню File → Export → Web → WAR file. После нажатия кнопки «Next» появится окно (рис. 2.6). В этом окне надо задать имя проекта и место размещения war-архива (он должен находиться в каталоге webapps сервера Apache Tomcat), затем нажать кнопку «Finish», после чего в каталоге webapps появится файл mvnweb1.war.

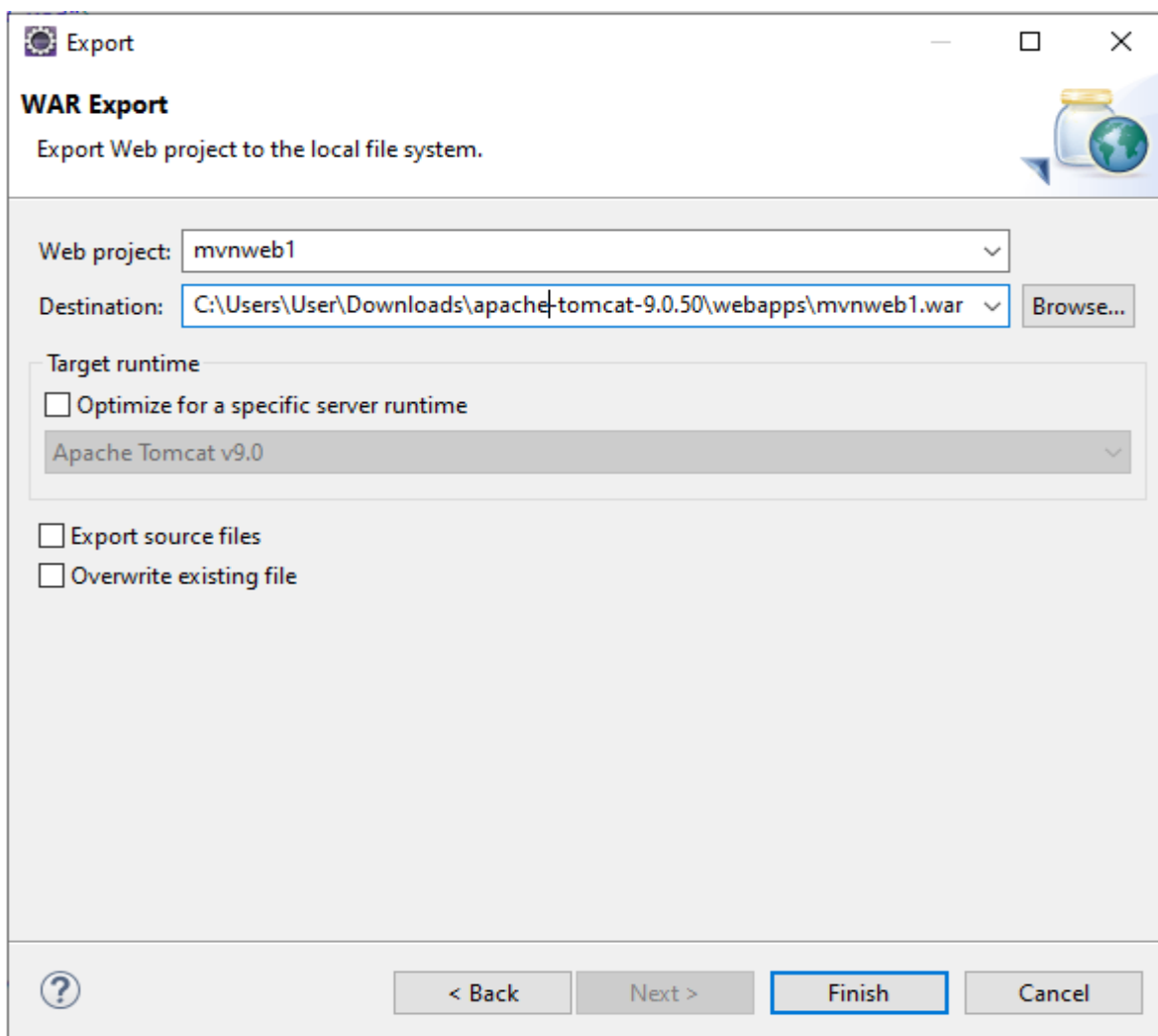


Рис. 2.6 Окно экспорта war-файла

Запустить веб-приложение можно в браузере, используя URL-адрес: `http: // <имя сервера>: <номер порта> /<имя проекта>/.` Например: `http://localhost:8080/mvnweb1/`.

Порядок выполнения лабораторной работы

1. Создайте Maven проект с архетипом `maven-archetype-webapp 1.4`.
2. Выполните сборку Web-приложения, используя команду `Maven install`.
3. Скопируйте `war`-файл проекта в каталог Web-сервера `Apache Tomcat`.
4. С помощью архиватора откройте и просмотрите `war`-файл.
5. Запустите Web-приложение в браузере.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Распечатку `pom.xml` файла проекта, в которую вставлены комментарии, описывающие зависимости и задачи плагинов.
2. Скриншот выполнения команды `Maven install`.
3. Структура и состав `war`- файла.
4. Скриншот экранной формы web-приложения.

Лабораторная работа № 3. ПОСТРОЕНИЕ WEB-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ СЕРВЛЕТОВ

Цель работы: знакомство с технологией построения Web-приложений на основе сервлетов.

Введение в технологию Java Servlet

Сервлет – это класс `Java`, предназначенный для динамического формирования ответа на запрос клиента по сети. Сервлеты часто называют Web-компонентами. Они выполняются в специальной среде исполнения (контейнере сервлетов), которую предоставляет Web-сервер. Спецификация сервлетов предполагает наличие в классе сервлета стандартных методов (инициализация, обработка запросов и завершение). Вызов этих методов осуществляется контейнером сервлетов.

Среда исполнения сервлетов обеспечивает некоторые полезные и экономящие время возможности, включая преобразование HTTP-запросов из сети в удобный для построения ответа объект и обратное преобразование объ-

екта в HTTP-ответ, который может быть послан обратно по сети. Характерной особенностью сервлетов является то, что они не требуют создания новых процессов при каждом новом запросе. Множество сервлетов выполняются параллельно в рамках одного процесса на сервере.

У каждого сервлета должно быть задано имя (servlet-name), по которому он идентифицируется на сервере. Это имя используется для задания свойств сервлета, в частности, для указания имени класса, в котором хранится программа сервлета (servlet-class), а также адреса, по которому можно обращаться к этому сервлету (url-pattern). По умолчанию адрес URL-запроса состоит из нескольких частей: имени компьютера, номера порта, каталога servlet, имени сервлета и списка параметров. Все эти свойства, а также имена (param-name) и значения (param-value) параметров инициализации хранятся на сервере в файле Web.xml, называемом дескриптором развертывания, в подкаталоге WEB-INF. Кроме параметров инициализации, которые сервлет читает из Web.xml-файла, в запросе могут присутствовать динамические параметры. Они указываются в запросе и имеют следующий вид:

Имя параметра = Значение параметра

Сервлеты размещаются в каталоге Webapps сервера в виде war-архивов (Web Application Archive). Обнаружив такой архив, Apache Tomcat его самостоятельно распаковывает и запускает Web-приложение. War-архив создается архиватором, входящим в состав Eclipse. После распаковки каждое приложение размещается в собственном одноименном каталоге с определенной вложенной структурой.

Жизненный цикл сервлета включает в себя следующие этапы:

- **Загрузку:** контейнер сервлетов загружает сервлет по прямому запросу или при своем запуске, используя дескриптор развертывания.
- **Инициализацию:** после загрузки контейнер вызывает метод init() и передает ему параметры инициализации. Метод init() должен быть выполнен перед тем, как контейнер начнет обрабатывать запросы. Вызывается он только один раз за весь жизненный цикл сервлета.
- **Обращение:** после успешной инициализации сервлет готов к обработке данных. Для каждого обращения к нему создается отдельный поток, управляемый самим контейнером, в котором вызывается метод service().
- **Уничтожение:** если сервлет больше не нужен, контейнер вызывает метод destroy(). Как и init(), он также может быть вызван лишь единожды.

Вызов метода прекращает отправку контейнером новых запросов к сервлету и высвобождает занятые им ресурсы.

Сервлеты взаимодействуют с клиентом в режиме запрос-ответ. Клиент обращается из браузера к Web-серверу, тот переадресует запрос сервлету, сервлет подготавливает ответ и передает его Web-серверу, который в свою очередь адресует его браузеру.

Настройка среды сервлета

Чтобы включить сервлет в Maven проект (WebApp1), созданный на основе архетипа `maven-archetype-webapp 1.4` (п. 2.1), необходимо в директорию `/src/main` проекта включить папку `java`. Эта операция выполняется через меню `File → New → Folder`. В появившемся окне (рис. 3.1), в поле `Folder name` надо вписать имя папки и нажать кнопку `Finish`.

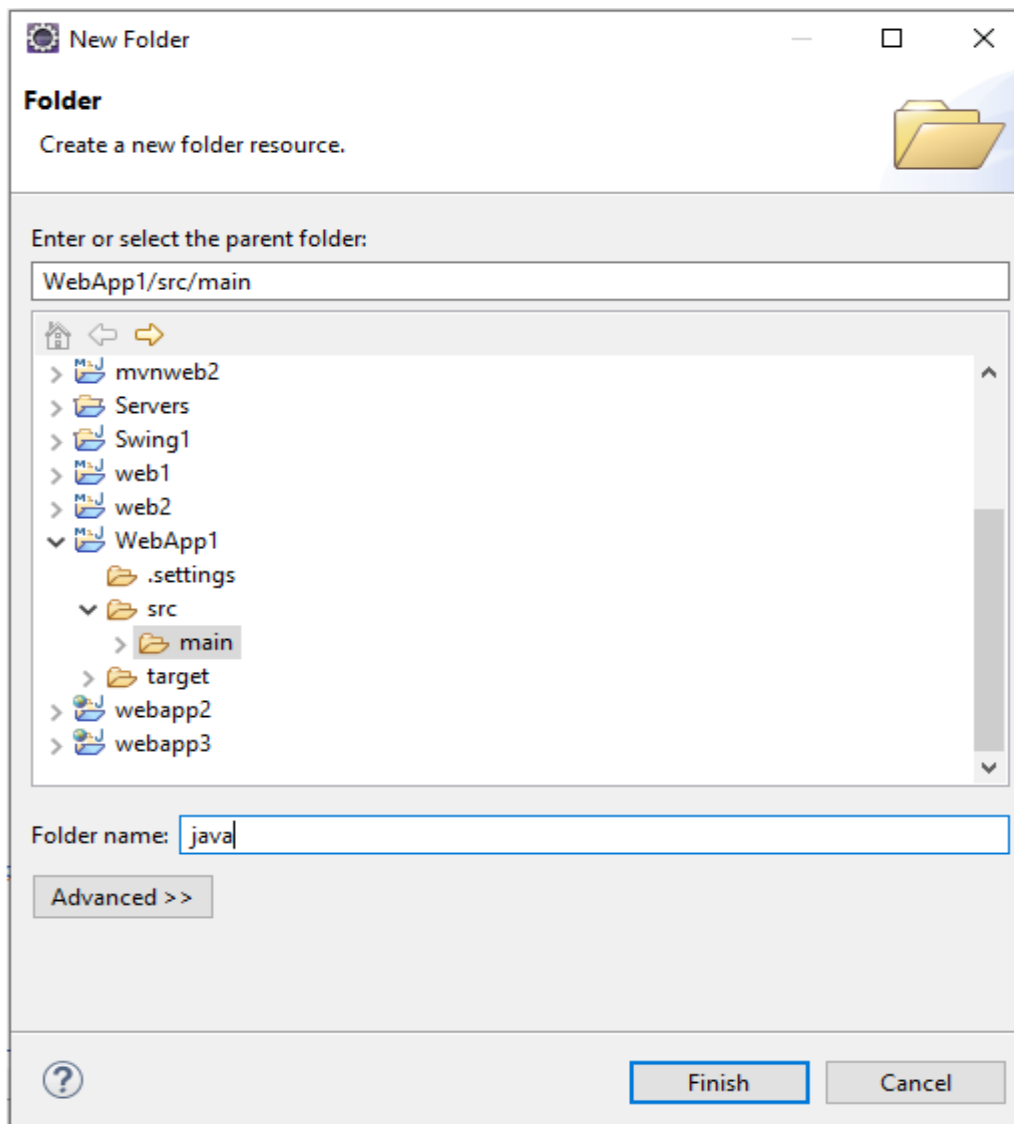


Рис. 3.1 Окно создание папки java

По умолчанию поддержка HttpServlet не включена в архетип, который использовался для создания Web-приложения. Поэтому необходимо поручить maven загрузить и обратиться к библиотекам поддержки HttpServlet. Это делается путем добавление следующей зависимости в pom.xml файл.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
</dependency>
```

Из проекта также надо удалить файл index.jsp.

Разработка сервлета

Для создания сервлета необходимо нажать правой клавишей мыши на имени проекта и выбрать пункты меню New → Other. В появившемся окне (рис. 3.3) выбрать пункты Web→Servlet и нажать кнопку Next.

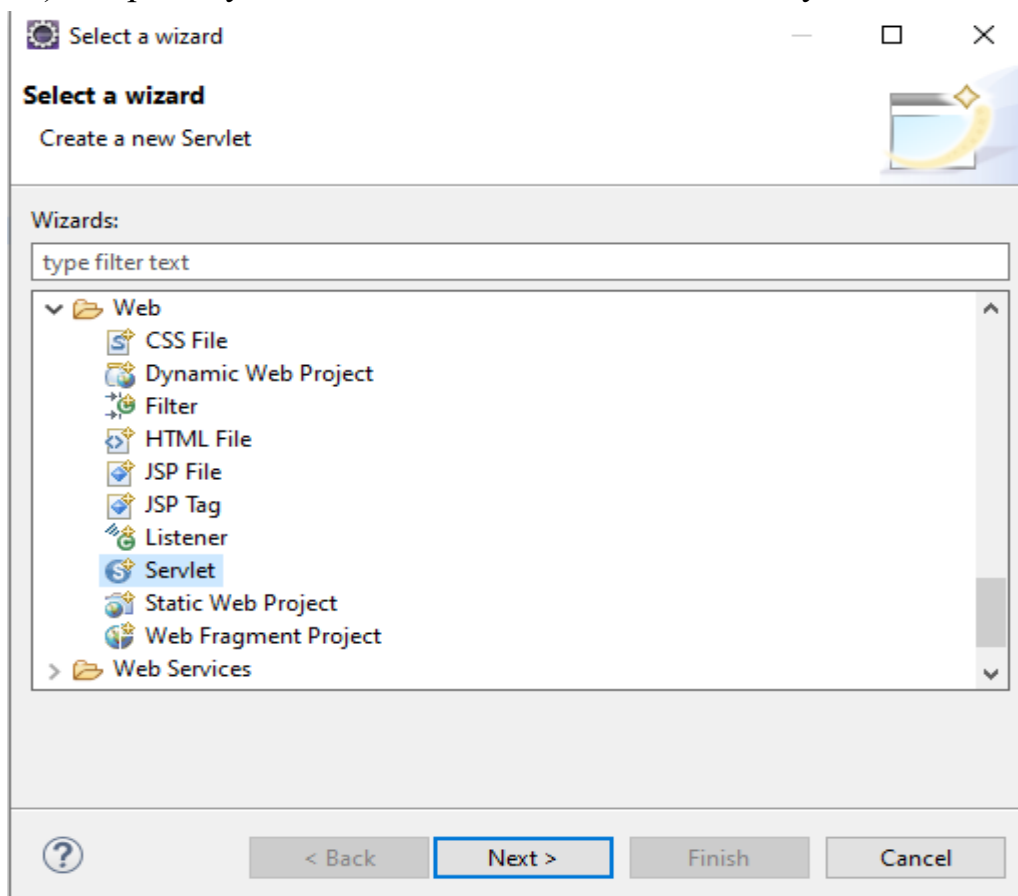


Рис 3.3 Окно выбора сервлета

Появится окно (рис. 3.4), в котором надо заполнить поля Class name (имя класса, например BooksList), Java package (имя пакета, например lab.web) и нажать кнопку «Finish».

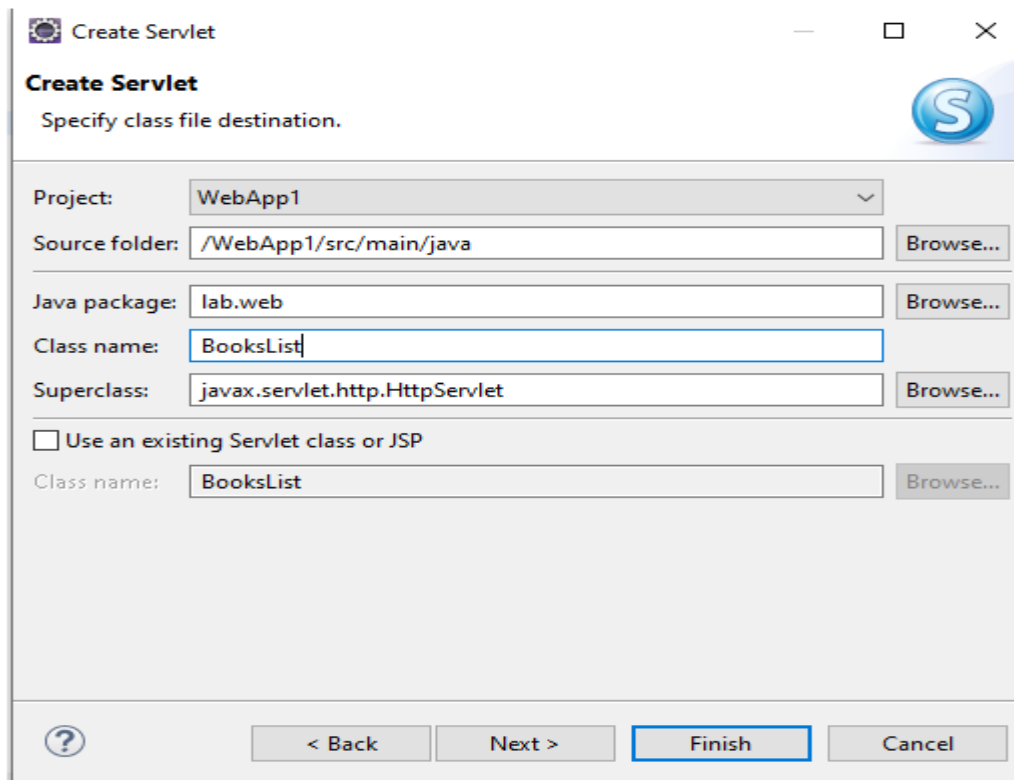


Рис. 3.4. Окно регистрации сервлета

После регистрации сервлета в файле web.xml проекта появится следующая секция:

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>BooksList</servlet-name>
    <display-name>BooksList</display-name>
    <description></description>
    <servlet-class>lab.web.BooksList</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>BooksList</servlet-name>
    <url-pattern>/BooksList</url-pattern>
  </servlet-mapping>
</web-app>
```

Эта секция делится на 2 части: объявление сервлета и отображение сервлета. В первой части указывается имя сервлета и пакет, в котором находится класс сервлета. Во второй части имя сервлета и URL-адрес обращения к сервлету. Один и тот же сервлет может быть сопоставлен с несколькими URL-адресами.

Для облегчения процесса разработки сервлета Eclipse генерирует следующий шаблонный класс:

```

package lab.web;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class BooksList
 */
public class BooksList extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public BooksList() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // TODO Auto-generated method stub
        response.getWriter().append("Served at: ").append(request.getContextPath());
    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        // TODO Auto-generated method stub
        doGet(request, response);
    }
}

```

В этом классе определены конструктор и 2 метода doGet и doPost, объявленных в абстрактном классе HttpServlet. Метод doGet вызывается при передаче HTTP-протоколом данных посредством запроса GET, а передача, осуществляемая при помощи запроса POST, обрабатывается методом doPost. Оба метода имеют 2 параметра, ассоциированных с запросом (request) и ответом (response). С помощью первого параметра можно получить данные от клиента, в том числе информацию о системном окружении клиента и данные,

которые клиент посылает на обработку в сервлет. Этот параметр представляет собой интерфейс `HttpServletRequest`, расширяющий базовый интерфейс `ServletRequest`. Подробное описание методов этих интерфейсов можно найти по адресу:

<http://docs.oracle.com/jaee/6/api/javax/servlet/ServletRequest.html>.

Второй параметр является интерфейсом `HttpServletResponse`, позволяющим сформировать поток вывода данных для формирования ответа. Ответ чаще всего оформляется в виде HTML-страницы, которая посылается обратно клиенту. Подробное описание методов этого интерфейса можно найти по адресу: <http://docs.oracle.com/jaee/6/api/javax/servlet/ServletResponse.html>.

Методы `doGet` и `doPost` используют одни и те же параметры, поэтому запрос клиента в сервлете может обрабатываться независимо от типа запроса. Для этого нужно описать отдельную операцию в сервлете, например `processRequest`, которая будет разбирать запрос и формировать ответ, используя соответственно методы интерфейсов `HttpServletRequest` и `HttpServletResponse`. Вызов этой операции нужно включить в тело методов `doGet` и `doPost`.

В шаблонном классе сервлета надо дополнить методы `doGet` и `doPost`, которые будут реализовывать его функциональность. Ниже приведено описание сервлета, который по запросу клиента выводит список книг читателя, имя которого указано в запросе.

```
package edu.etu.Web;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
// библиотечный класс для работы с потоками вывода
import java.io.PrintWriter;
/**
 * Servlet implementation class BooksList
 */
// @WebServlet("/BooksList")
public class BooksList extends HttpServlet {
    private static final long serialVersionUID = 1L;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public BooksList() {
        super();
        // TODO Auto-generated constructor stub
    }
}
```

```

    }
    /**
    * Processes requests for both HTTP GET and POST
methods.
    *
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs
    * @throws IOException if an I/O error occurs
    */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        // Задание типа кодировки для параметров запроса
        request.setCharacterEncoding("utf-8");
        // Чтение параметра name из запроса
        String name = request.getParameter("name");
        // Задание типа содержимого для ответа (в том числе кодировки)
        response.setContentType("text/html;charset=UTF-8");
        // Получение потока для вывода ответа
        PrintWriter out = response.getWriter();
        try {
            // Создание HTML-страницы
            out.println("<html>");
            out.println("<head><title>Список книг</title></head>");
            out.println("<body>");
            out.println("<h1>Список книг читателя " + (name != null ? name :
"без имени") + "</h1>");
            out.println("<table border='1'>");
            out.println("<tr><td><b>Автор книги</b></td><td><b>Название книги
</b></td><td><b>Прочитал</b></td></tr>");
            out.println("<tr><td>Булгаков</td><td>Мастер и Маргарита
</td><td>Да</td></tr>");
            out.println("<tr><td>Пелевин</td><td>Чапаев и пустота
</td><td>Нет</td></tr>");
            out.println("</table>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            // Закрытие потока вывода
            out.close();
        }
    }
    /**
    * Handles the HTTP
    * GET method.
    *
    * @param request servlet request
    * @param response servlet response
    * @throws ServletException if a servlet-specific error occurs

```

```

    * @throws IOException if an I/O error occurs
    */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
    /**
     * Handles the HTTP
     * <code>POST</code> method.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
}

```

В этом описании при формировании ответа использовался метод `println`, который выводит в выходной поток строку HTML-кода. После завершения вывода этот поток необходимо закрыть. При формировании HTML-страницы использовались следующие теги:

`<title>` – название страницы;

`<h1>` – вывод заголовка;

`<table border='1'>` – создание таблицы с рамкой толщиной в 1 пиксель;

`<tr><td>` – вывод ячейки таблицы.

Подробное описание тегов языка HTML можно найти по адресу:

<http://www.Webremeslo.ru/spravka/spravka.html>

При выводе заголовка анализируется параметр `name`. Если он указан в запросе, то в заголовке печатается его значение, в противном случае выводится строка «без имени».

В описании класса показана возможность использования аннотации `@WebServlet("/BooksList")` для регистрации сервлета. Она применяется в том случае, если регистрация сервлета не была проведена в `web.xml` файле.

Запуск сервлета

Перед запуском сервлета необходимо выполнить команду `Maven install`. По этой команде в каталоге `target` будет создан файл `WebApp1.war`.

Этот файл надо скопировать в каталог webapps и запустить сервер Tomcat. Для запуска сервлета на локальном компьютере при использовании стандартного порта подключения 8080 необходимо в браузере задать адрес, в котором кроме этих данных надо указать имя каталога WebApp1 и имя сервлета. Например, для описанного в 3.3 сервлета такой адрес будет следующим:

<http://localhost:8080/WebApp1/BooksList>

Если требуется в запросе передать дополнительные динамические параметры, то после адреса надо поставить знак «?» и задать имя параметра и его значение, например:

<http://localhost:8080/WebApp1/BooksList?name=Иванов И.И.>

По этому запросу в браузере отобразится следующая HTML-страница (рис. 3.5).

Список книг читателя Иванов И.И.

Автор книги	Название книги	Прочитал
Булгаков	Мастер и Маргарита	Да
Пелевин	Чапаев и пустота	Нет

Рис.3.5. Результат работы сервлета

При отладке сервлета приходится несколько раз изменять его код. После таких изменений необходимо заново создавать war-файл, предварительно удалив из каталога webapps старый архив и разархивированный его каталог. Следует также иметь в виду, что браузер запоминает в кеш-памяти открываемые HTML-страницы и при обращении к сервлету с предыдущим запросом он отобразит старую HTML-страницу. Поэтому перед запуском обновленного сервлета необходимо очистить кеш-память браузера, нажав комбинацию клавиш Ctrl+H.

Документирование проекта

Для создания документации проекта необходимо в файл pom.xml добавить плагин maven-javadoc-plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>3.1.1</version>
  <executions>
    <execution>
      <id>attach-javadocs</id>
      <goals>
```



```
<goal>javadoc</goal>
</goals>
</execution>
</executions>
</plugin>
```

Запустить этот плагин можно из командной строки `mvn javadoc:javadoc` текущего проекта или из Eclipse по команде Maven build с указанием в поле Goals значения `javadoc:javadoc` (рис. 2.5). Документация сгенерируется в *target/site/apidocs* каталоге текущего проекта.

Порядок выполнения лабораторной работы

1. Выберите предметную область для лабораторной работы и разработайте для нее Web-приложение (например, можно взять предметную область из курсового проекта по объектно-ориентированному программированию на языке Java).
2. Создайте Maven проект для Web-приложения.
3. Разработайте HTML-страницу и сервлет для Web-приложения. Сервлет должен принимать параметры инициализации и один или более динамических параметров.
4. По команде Maven install создайте war-архив и поместите его на сервере Apache Tomcat.
5. Запустите Web-сервер и обратитесь к сервлету из браузера, проверьте правильность отображения HTML-страницы и работу сервлета.
6. Сгенерируйте документацию для сервлета с помощью команды Maven build и просмотрите ее в браузере.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Скриншоты, иллюстрирующие работу Web-приложения.
2. Распечатки исходного текста сервлета, файлов `web.xml` и `pom.xml`.
3. Текст документации сервлета, сгенерированный Javadoc.

Лабораторная работа № 4. ИНТЕРНАЦИОНАЛИЗАЦИЯ WEB-ПРИЛОЖЕНИЙ

Цель работы: знакомство со способами отображения данных на различных языках при использовании файлов ресурсов.

Java-интернационализация приложения

Интернационализация – это написание приложения, работающего в различных языковых окружениях. Основной целью интернационализации является создание многоязычного пользовательского интерфейса и поддерживающей его инфраструктуры. Объектами интернационализации приложения являются:

- Сообщения – представление всего видимого текста (текст сообщения, текст ошибки, заголовки компонентов пользовательского интерфейса, приглашения и т. д.) на языке, соответствующем контексту региональной настройки исполнения.
- Форматы – использование правильных, зависящих от региональной настройки форматов даты, времени и числовых значений.
- Календари и временные зоны – использование календаря, соответствующего региональной настройке исполнения приложения.
- Строковые данные – использование соответствующей политики строкового сравнения на основе языка региональной настройки.
- Общие свойства пользовательского интерфейса, чувствительные к местной специфике изображения, значки и цвета – использование изображений и цветов, которые представляют собой многозначную символику для местных пользователей.

Процесс интернационализации можно разбить на 2 основных этапа: подготовка программы к адаптации (локализации) к конкретному языку и определение ресурсов, необходимых для локализации. На первом этапе необходимо решить следующие задачи:

- определить элементы пользовательского интерфейса, требующие локализации;
- исследовать влияние изменения длины строк на работоспособность программы;
- изучить последствия использования в ресурсах символов из другого языка.

Основным классом, отвечающим за представление данных на разных языках, является класс `java.util.Locale`. Определено 2 варианта конструкторов в классе `Locale`:

`Locale (String language, String country);`

`Locale (String language, String country, String variant).`

В первых двух параметрах обоих конструкторов в кодировке ISO задаются язык и страна, на которые настраивается приложение. Язык описывается двухбуквенным кодом в нижнем регистре, страна обозначается двухбуквенным кодом и записывается в верхнем регистре. Список поддерживаемых языков и стран можно получить с помощью вызова статических методов `Locale.getISOLanguages()` и `Locale.getISOCountries()`. Во втором варианте конструктора указан строковый параметр `variant`, который не имеет какого-то определенного смысла и предназначен для указания некой дополнительной информации, например об операционной системе. Примеры конструкторов:

```
Locale lc1 = new Locale ("ru","RU");  
Locale lc2 = new Locale ("en","US","WINDOWS").
```

Текущее языковое окружение описывается понятием локаль (место действия). Текущая локаль по умолчанию устанавливается исходя из установок ОС, получить ее можно обратившись к статическому методу `Locale.getDefault()` или `request.getLocale()`, а переопределить – `Locale.setDefault(locale)` или `request.setLocale(locale)`. Однако следует помнить, что эти методы воздействует только на Java-программу, а не меняет настройки в ОС. Для наиболее употребимых локалей можно использовать константы. Например, константа `Locale.ENGLISH` будет задавать английский язык, а константа `Locale.GERMAN` – немецкий.

Создание и загрузка файла ресурсов

Для хранения данных (ресурсов), зависящих от локали, используют класс `ResourceBundle`, который хранится в библиотеки `java.util.ResourceBundle`. Этот класс абстрактный, поэтому получить его экземпляр можно исключительно с помощью статического метода `getBundle`, имеющего несколько перегруженных реализаций. В данной лабораторной работе будет использован вариант, в котором для хранения ресурсов служит файл с расширением `properties`.

Концептуально каждый `ResourceBundle` является набором соответствующих файлов, разделенных одним и тем же именем. Их количество определяется числом локалей, которые будут участвовать в процессе локализации приложения. В каждом имени файла символы, следующие за базовым именем и разделенные символом «`_`», показывают код языка, код страны и вариант, указанные в локали. Например, для базового имени `book` могут быть определены следующие имена файлов ресурсов:

```
Book.properties
Book_ru.properties
Book_ru_RU.properties
Book_ru_RU_UNIX.properties
Book_ru_RU_WINDOWS.properties
Book_en.properties
Book_en_US.properties.
```

Чтобы загрузить в приложение определенный ресурс, нужно задать локаль и вызвать метод `getBundle`:

```
Locale currentLocale = new Locale ("ru", "RU", "WINDOWS");
ResourceBundle introLabels = ResourceBundle .getBundle("Book", currentLocale);
```

Если метод `getBundle` не может найти соответствия в заданном списке файлов, то он выбрасывает исключение `MissingResourceException`. Чтобы обойти выбрасывание исключения, необходимо всегда создавать базовый файл без суффиксов (в данном примере это `Book.properties`), который будет загружаться в приложение, когда не будет найден файл ресурсов для указанной локали.

Ресурсный файл создается как обычный текстовый файл, содержащий перечень строк вида «ключ=значение». Ключ – это имя, которое может быть использовано программой для получения значения. Как в ключах, так и в значениях допускаются пробелы. Текст в файле должен быть записан в кодировке `Unicode`. Например, файлы ресурсов с русским и английским переводом компонентов интерфейса приложения, отображающего список читателя (см. лабораторную работу № 3), будут содержать следующую информацию:

```
Файл Book.properties
title=Список книг
author=Автор книги
book.title=Название книги
read=Прочитал
Файл Book_en.properties
title=Book list
author=Author name
book.title=Book title
read=Have read
```

Оба этих файла надо поместить в предварительно созданный каталог проекта `/src/main/resources`.

Локализация компонентов интерфейса и кода

Для локализации компонентов интерфейса и строк сообщений, появляющихся в коде приложения, необходимо заменить жестко запрограммиро-

ванные названия и описания выражениями, которые ссылаются на переведенные названия и строки в локализованном файле ресурсов. Если необходимо переключение локализаций, то целесообразно информацию о нужной локализации передавать в качестве параметра сервлету или хранить эту информацию в сессии, так как она требуется всем страницам приложения в рамках сессии.

Ниже приведено описание сервлета, который по запросу клиента выводит список книг читателя на русском или английском языке. Язык задается параметром lang, который может принимать значения «ru» или «en».

<http://localhost:8080/WebApp2/BooksList?lang=ru>

<http://localhost:8080/WebApp2/BooksList?lang=en>

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // Чтение параметров из строки
    String lang = request.getParameter("lang");
    if(lang == null) {
        response.sendError(HttpServletResponse.SC_NOT_ACCEPTABLE,
"Ожидался параметр lang");
        return;
    }
    if(!"en".equalsIgnoreCase(lang) && !"ru".equalsIgnoreCase(lang)) {
        response.sendError(HttpServletResponse.SC_NOT_ACCEPTABLE,
"Параметр lang может принимать значения en или ru");
        return;
    }
    // Задание типа содержимого для ответа (в том числе кодировки)
    response.setContentType("text/html;charset=UTF-8");
    // Файлы ресурсов book.properties, book_en.properties и book_ru.properties
    // Установка локализации в соответствии с выбором пользователя
    ResourceBundle res = ResourceBundle.getBundle(
"/Book", "en".equalsIgnoreCase(lang) ? Locale.ENGLISH : Locale.getDefault());
    // Получение потока для вывода ответа
    PrintWriter out = response.getWriter();
    try {
        // Создание HTML-страницы
        out.println("<html>");
        out.println("<head><title>");
        // Вывод строки с учетом локализации
        String src1=res.getString("title");
        out.print(new String(src1.getBytes("ISO-8859-1"),"UTF-8" ));
        out.println("</title></head>");
        out.println("<body>");
        out.println("<h1>");
        src1=res.getString("title");
        out.print(new String(src1.getBytes("ISO-8859-1"),"UTF-8" ));
        out.println("</h1>");
```

```

        out.println("<table border='1'>");
        out.println("<tr><td><b>");
        src1=res.getString("author");
        out.print(new String(src1.getBytes("ISO-8859-1"),"UTF-8" ));
        out.println("</b></td><td><b>");
        src1=res.getString("book.title");
        out.print(new String(src1.getBytes("ISO-8859-1"),"UTF-8" ));
        out.println("</b></td><td><b>");
        src1=res.getString("read");
        out.print(new String(src1.getBytes("ISO-8859-1"),"UTF-8" ));
        out.println("</b></td></tr>");
        out.println("<tr><td>Булгаков</td><td>Мастер и Маргарита </td><td>Да</td></tr>");
        out.println("<tr><td>Пелевин</td><td>Чапаев и пустота</td><td>Нет</td></tr>");
        out.println("</table>");
        out.println("</body>");
        out.println("</html>");
    } finally {
        // Закрытие потока вывода
        out.close();
    }
}

```

В приведенном примере загрузка файла ресурсов в соответствии с выбором пользователя осуществляется вызовом метода `Locale.getDefault()`, который хранится в библиотеке `java.util.Locale`. Если параметр `lang` имеет значение «en», то будет загружен ресурс `Book_en.properties`, а при значении «ru» – `Book.properties`, который задает русский язык и является ресурсом по умолчанию.

В HTML-коде сервлета требуется замена текстовых констант на вызов метода `res.getString("ключ")`, который будет подставлять текст, соответствующий заданному ключу. Этот метод возвращает строку, используя кодировку ISO-8859-1. Для корректного вывода русских букв надо перекодировать строку в формат UTF-8. Это преобразование выполняет метод `getBytes("ISO-8859-1"),"UTF-8")`, который кодирует данную строку в последовательность байт с указанной кодировкой.

Порядок выполнения лабораторной работы

1. Проанализируйте разработанное в лабораторной работе № 3 приложение и определите, для каких элементов пользовательского интерфейса можно провести локализацию.
2. Создайте новый Maven web-проект, аналогичный лабораторной работе № 3, и добавьте в него каталог ресурсов.

3. Создайте файлы ресурсов для английского и русского языков и включите их в каталог ресурсов.
4. Внесите изменения в сервлет приложения для возможности переключения интерфейса на английский и русский языки.
5. Запустите приложения в двух режимах и снимите скриншоты интерфейсов на английском и русском языках.
6. Сгенерируйте документацию проекта с помощью команды Maven build и просмотрите ее в браузере.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Скриншоты, иллюстрирующие работу Web-приложения.
2. Распечатки исходного текста сервлета и файлов ресурсов.
3. Текст документации проекта, сгенерированный Javadoc.

Лабораторная работа № 5. ПОСТРОЕНИЕ Web-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ JSP

Цель работы: знакомство с технологией построения Web-приложений на основе JSP.

Модель JSP-страницы

Технология JSP (JavaServer Pages) является расширением технологии сервлетов. Основное назначение JSP-страниц – упростить создание и управление динамическим содержанием Web-приложения. Программист создает JSP-страницу, где могут содержаться фрагменты двух типов: статические данные в виде HTML- или XML-кода и JSP-элементы, которые задают динамическую составляющую страницы, генерируемую при помощи Java-кода. JSP-страница преобразуется в обычный сервлет со статическим HTML-кодом, который направляется в поток вывода, связанный с методом сервлета `service`. Одно из преимуществ технологии JSP над сервлетами состоит в том, что процесс построения сервлета выполняется автоматически. Когда страница JSP в первый раз запрашивается пользователем или она была изменена, то сначала выполняется ее предобработка JSP-парсером, а затем генерируется класс реализации JSP-страницы. При этом парсер транслирует операторы Java в метод `_jspService()`, необходимый для выполнения кода, записанного в JSP-странице. Этот код загружается в контейнер сервлета, после чего вызываются методы `init()` и `_jspService()`.

На JSP-странице кроме HTML-кода могут присутствовать еще 3 основных типа JSP-конструкций: элементы скриптов, директивы и действия. Элементы скриптов позволяют задать код на языке Java, который впоследствии станет частью сервлета, директивы дают возможность управлять всей структурой сервлета, а действия служат для задания существующих используемых компонентов, а также для контроля поведения движка JSP. Перечень основных конструкций и их синтаксис приведены в таблице.

Тип конструкции	JSP-элемент	Синтаксис	Описание
Элементы скриптов	Выражения	<code><%= Выражение на Java %></code>	Java-выражение вычисляется при запросе, конвертируется в строку и вставляется в страницу
	Скриптлеты	<code><% Код на Java %></code>	Java-код вставляется в метод service сервлета
	Объявления	<code><%! Код на Java %></code>	Java-код вставляется в тело класса сервлета вне метода service
Директивы	page	<code><%@ page атрибут="значение" %></code>	Задаёт значения атрибутов для JSP-страницы, например импорт пакетов (import), сохранение информации о клиенте (session)
	include	<code><%@ include file.jsp = "относительный url" %></code>	Включает JSP-файл в процессе трансляции JSP-страницы
	taglib	<code><%@ taglib uri="URI к библиотеке знаков" prefix="префикс знака" %></code>	Присоединяет библиотеку знаков для определения собственных тегов
Действия	include	<code><jsp:include page="относительный URL" flush="true" /></code>	Подключает ответ от JSP-файла при вызове JSP-страницы
	forward	<code><jsp:forward page = "относительный URL"/></code>	Передаёт запрос к другой странице

Окончание таблицы

Тип конструкции	JSP-элемент	Синтаксис	Описание
Комментарии	JSP-комментарий	<code><%-- комментарий --%></code>	Игнорируется транслятором JSP
	HTML-комментарий	<code><!-- комментарий --></code>	Передается в конечный HTML

В JSP-выражениях и скриплетах доступны predefined переменные (объекты) `request`, `response`, `out`, `session`, `application`, `config`, `pageContext` и `page`, которые не надо декларировать и можно использовать для вызова соответствующих методов. Каждый такой неявный объект имеет тип класса или интерфейса, определенный в основной технологии Java или в пакете Java Servlet API.

Встроенный объект `request` является объектом класса `javax.servlet.HttpServletRequest` и предназначен для получения информации о клиентском запросе. Методы этого объекта описаны по адресу <http://spec-zone.ru/RU/Java/EE/6.0.1/docs/api/javax/servlet/http/HttpServletRequest.html>.

С помощью переменной `response` формируется ответ для клиента. Она является объектом класса `javax.servlet.HttpServletResponse`, методы которого описаны по адресу <http://spec-zone.ru/RU/Java/EE/6.0.1/docs/api/javax/servlet/http/HttpServletResponse.html>. Эти методы позволяют дополнить выходной поток данных служебной информацией благодаря чтению и установке заголовка ответа, перенаправлению данных, кодированию URL, добавлению данных о клиенте (`cookie`).

Переменная `out` является объектом класса `Print Writer`. Она используется в скриплетах для передачи сообщений в браузер клиента, формируя выходной поток с помощью методов, описанных по адресу <http://spec-zone.ru/RU/Java/Docs/8/api/java/io/PrintWriter.html>.

Отражением клиентской сессии (сеансом связи с конкретным клиентом) является объект `session`. Он является экземпляром класса `javax.servlet.http.HttpSession` и создается контейнером для идентификации

клиента, а также хранения персональных данных. Он доступен для всех JSP-страниц, обрабатывающих запросы, связанные с определенным клиентом. Сервер различает сессии с помощью маркера, который передается пользователю. Маркер хранится в cookies браузера до конца сессии. Используя методы родительского класса, описанные по адресу <http://spec-zone.ru/RU/Java/EE/6.0.1/docs/api/javax/servlet/http/HttpSession.html>, можно получить свойства сессии, время ее создания, добавлять данные в сессию, извлекать и удалять их. Данные в объекте session хранятся в виде пары типа «ключ – объект» на протяжении всего периода работы с приложением. При этом следует иметь в виду, что отследить момент, когда пользователь перестает работать с приложением, не всегда возможно. Поэтому в настройках Web-сервера устанавливается некоторое предельное время существования объекта session после получения последнего запроса. По истечении этого времени объект session уничтожается.

Иногда серверу необходимо хранить переменные на уровне Web-приложения, чтобы каждый клиент мог использовать копии этих данных и манипулировать ими. Для этого служит встроенный объект application, который не привязан к какой-либо отдельной JSP-странице или сеансу связи, а доступен со всех JSP-страниц данного приложения. Объект application принадлежит классу ServletContext (<http://spec-zone.ru/RU/Java/EE/6.0.1/docs/api/javax/servlet/ServletContext.html>), и с помощью его методов можно получить внутренние сведения об исполняемом сервлете и хранить данные подобно объекту session.

Встроенный объект pageContext является объектом класса javax.servlet.jsp.PageContext (<http://spec-zone.ru/RU/Java/EE/6.0.1/docs/api/javax/servlet/jsp/PageContext.html>) и обеспечивает доступ ко всему пространству имен, связанных с одной JSP-страницей. Основная идея использования PageContext и его унаследованных методов getAttributeNamesInScope(), getAttributesScope(), findAttribute(), getAttribute() заключается в организации доступа к параметрам данной страницы при помощи других объектов для совместной работы.

Обработка исключительных ситуаций в JSP-страницах

Существует несколько типов ошибочных ситуаций, возникающих на различных этапах жизни JSP-страницы. К первому типу можно отнести синтаксические ошибки в конструкциях, используемых в JSP-странице. Обычно

эти ошибки выявляются редактором среды разработки Eclipse, способным выполнять синтаксический контроль в момент набора страницы. Синтаксические ошибки также могут быть обнаружены JSP-контейнером в момент трансляции и компилирования JSP-страницы. Обычно ошибки данного типа возникают в Java-коде, в частности при работе с директивами и встроенными объектами. Если ошибка возникает при трансляции страницы (например, транслятор находит элемент JSP с неправильным форматом), сервер возвращает ошибку `ParseException`, и исходный файл класса сервлета будет пустым или незаконченным. Последняя незаконченная строка дает указатель на неправильный элемент JSP. Если ошибка возникает при компиляции страницы (например, синтаксическая ошибка в скриптлете), сервер возвращает ошибку `JasperException` и сообщение, которое включает в себя имя сервлета JSP-страницы и строку, в которой произошла ошибка.

Ошибки, возникающие на этапе прогона JSP-страницы, должны обрабатываться в теле класса сервлета, используя механизм исключений языка программирования Java. Любые неотловленные исключения, вызываемые в теле класса сервлета, приводят к перенаправлению клиентского запроса и неотловленного исключения по URL `errorPage`, специфицированному этой JSP-страницей или выполнением действий по умолчанию, если ничего не специфицировано. В интерфейс `HttpServletResponse` входят константы для обозначения всех кодов состояний и ошибок протокола HTTP. Их можно разделить на следующие категории:

- серия 100-199 – информационные коды, процесс продолжается;
- серия 200-299 – коды указывают, что процесс был успешным;
- серия 300-399 – переадресация, для выполнения запроса требуются дальнейшие действия;
- серия 400-499 – ошибка клиента, в запросе содержатся синтаксические ошибки или запрос не может быть выполнен;
- серия 500-599 – ошибка сервера, сервер отказывается выполнять заведомо корректный запрос.

Например, сервер выдаст следующую ошибку, связанную с неправильным указанием пути, или при обращении к JSP-странице, которая не существует: «Error 404 - Not Found». С полным описанием перечня кодов HTTP-ошибок можно ознакомиться по адресу <http://bourabai.kz/xml/app1.htm>.

При создании JSP-приложения можно включить собственный обработчик исключительных ситуаций на языке Java, используя блок try...catch, или же обработать ситуацию с помощью JSP-страницы. Подключить JSP-страницу для обработки ошибки можно двумя способами. Первый способ использует директиву `<%@ page errorPage="file_name" %>`, которая записывается в начале JSP-страницы и задает имя файла обработчика ошибок для данной страницы. Например, если в качестве такого обработчика будет использован JSP-файл с именем `error`, то директива объявляется так:

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

Этот способ предполагает, что все ошибочные ситуации, возникающие в процессе запуска страницы, будут обрабатываться только одной JSP-страницей, имя которой указано в директиве `page`.

Второй способ обработки ошибки позволяет задать обработчики для разных страниц Web-приложения. Для этого надо включить элемент `error-page` в файл `Web.xml` проекта. Этот элемент задает соответствие между типом ошибки и JSP-страницей, которая их обрабатывает:

```
<error-page>
<exception-type> Тип исключения</exception-type>
<location>Имя JSP- файла</location>
</error-page>
```

Например, для обработки всех исключений ввода-вывода JSP-страницей с именем `error1`, размещенной на сервере в корневом каталоге приложения, элемент `error-page` будет иметь следующий вид:

```
<error-page>
<exception-type>java.io.IOException</exception-type>
<location>/error1.jsp </location>
</error-page>
```

Если требуется обрабатывать HTTP-ошибки, например с кодом 404, то в файл `Web.xml` нужно добавить следующие строки:

```
<error-page>
<error-code>404</error-code>
<location>/error2.jsp </location>
</error-page>
```

В JSP-файле обработчика ошибок необходимо объявить директиву `<%@ Page %>` и атрибуту `isErrorPage` присвоить значение `true`. Эта директива указывает, что данная страница является обработчиком ошибок и ей доступны объекты исключения. Например, JSP-страница, осуществляющая

вывод сообщений для стандартных исключительных ситуаций, будет выглядеть следующим образом:

```
<%@ page isErrorPage="true" %>
<STML><BODY><H1>Error Page</H1>
<H2>Received the exception:
<FONT color=red>
<% exception.toString() %>
</FONT></H2></BODY></HTML>
```

Создание JSP-страницы

Разработка Web-приложения на основе JSP-технологии в среде Eclipse начинается с открытия проекта (см. 2.1). Архетип этого проекта предполагает создание в каталоге webapp файла index.jsp, который можно использовать для хранения собственной jsp-страницы, либо открыть другой файл с помощью меню New→JSP File. В появившемся окне надо указать имя файла (Рис. 5.1 в данном примере это BookList.jsp) и нажать кнопку Next.

После нажатия кнопки «Next» появится окно, в котором надо выбрать тип шаблона JSP-страницы (рис. 5.2).

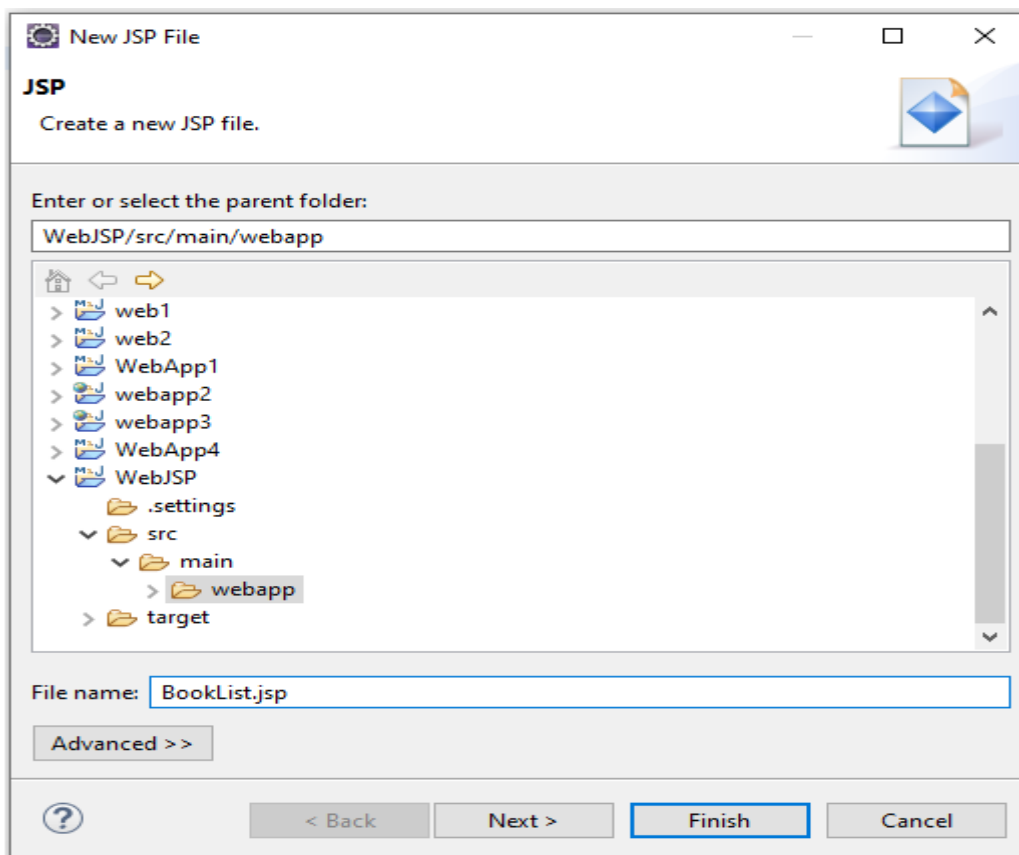


Рис. 5.1 Окно создания jsp-файла

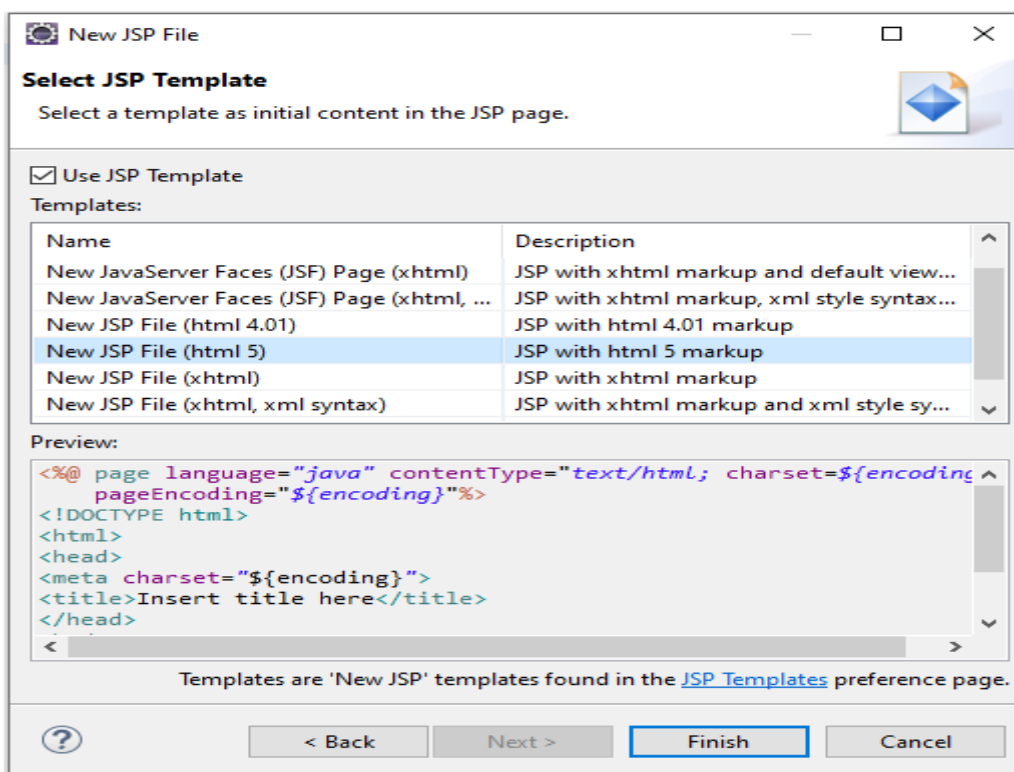


Рис. 5.2. Окно выбора шаблона JSP-страницы

Рекомендуется выбрать тип шаблона «New JSP-File (html 5)», который задаст стандартные директивы и позволит использовать языки Java и HTML 5.0 при создании JSP-страницы. После нажатия кнопки «Finish» в соответствии с выбранным шаблоном Eclipse создаст новую JSP-страницу и откроет ее в JSP-редакторе. Этот текст надо дополнить кодом согласно задаче, которую должно решать Web-приложение.

При написании JSP-страницы рекомендуется придерживаться следующей структуры:

1. Начальный комментарий.
2. Директивы JSP-страницы.
3. Директивы библиотеки тегов (опционально).
4. Различные JSP-объявления (опционально).
5. HTML- и JSP-коды.

В качестве примера ниже приведено описание трех JSP-страниц. В первой странице (файл BookList.jsp) выводится заголовок таблицы и осуществляется проверка параметра name (имя читателя), во второй (файл ListData.jsp) – выводится тело таблицы, причем данная страница включена в первую с помощью директивы include, и в третьей (файл ErrorManager.jsp) –

осуществляется обработка исключительной ситуации, если не задано значение параметра name.

Все файлы должны быть созданы в редакторе Eclipse и помещены в каталог WEB-INF проекта. В шаблон JSP-страницы включена директива page, в которой необходимо выставить кодировку UTF-8 самой страницы. Чтобы входные параметры запроса читались также в кодировке UTF-8, необходимо воспользоваться методом `request.setCharacterEncoding("UTF-8")`.

Файл BookList.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title> Список книг</title>
</head>
<body>
    <% request.setCharacterEncoding("UTF-8");
        String name = request.getParameter("name");
        RequestDispatcher dispatcher;
        if (name == null)
        { dispatcher = getServletContext().getRequestDispatcher("/ErrorManager.jsp");
          dispatcher.forward(request, response); }
    %>
    <h1> Список книг читателя <%=name%></h1>
    <%@include file="ListData.jsp"%>
</body>
</html>
```

Файл ListData.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<table border='1'>
    <tr>
        <td><b>Автор книги</b></td>
        <td><b>Название книги</b></td>
        <td><b>Прочитал</b></td>
    </tr>
    <tr>
        <td>Булгаков</td>
        <td>Мастер и Маргарита</td>
        <td>Да</td>
    </tr>
    <tr>
        <td>Пелевин</td>
        <td>Чапаев и пустота</td>
```

```

        <td>Нет</td>
    </tr>
</table>

```

Файл ErrorManager.jsp:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Обработка ошибки</title>
    </head>
    <body>
        <h3>Ошибка</h3>
        Необходимо ввести имя после URL в формате "?name=Имя"<br>
    </body>
</html>

```

Настройка и запуск JSP-страницы

Настройка JSP-страницы предполагает выполнение следующих шагов:

1. Добавление в pom.xml файл проекта следующих зависимостей:

```

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>5.0.0</version>
</dependency>

```

2. Создание war-файла и его развертывание на Web-сервере.

3. Запрос к стартовой JSP-странице.

Запрос к стартовой JSP-странице может быть осуществлен с указанием полного URL (<http://localhost:8080/WebJSP/BookList.jsp?name=Иванов И.И.>), где прописаны имя и точный путь расположения JSP-файла на сервере, или URL, задающего только каталог, а не конкретное имя файла (<http://localhost:8080/WebJSP/?name=Иванов И.И.>). Во втором случае контейнер сервлетов будет искать в дескрипторе развертывания элемент welcome-file-list со списком начальных файлов и анализировать их в порядке перечисления, пока указанный в списке файл не будет найден в корневом каталоге Web-приложения. Этот файл запускается, и ответ возвращается клиенту. Если ни один из файлов списка не найден в корневом каталоге Web-

приложения, то контейнер сервлетов возвращает клиенту ошибку. При использовании второго варианта запроса необходимо в файл Web.xml проекта в секции <web-app> и <welcome-file-list> поместить элемент <welcome-file> «имя файла» </welcome-file>, в котором имя файла будет соответствовать стартовой JSP-странице. Для рассмотренного ранее примера такой элемент будет иметь следующий вид:

```
<web-app>  
  <display-name>Archetype Created Web Application</display-name>  
  <welcome-file-list>  
    <welcome-file>BookList.jsp</welcome-file>  
  </welcome-file-list>  
</web-app>
```

При запуске приложения по адресу <http://localhost:8080/WebApp2?name=Петров П.П.> в браузере отобразится следующая HTML-страница:

Список книг читателя Петров П.П.

Автор книги	Название книги	Прочитал
Булгаков	Мастер и Маргарита	Да
Пелевин	Чапаев и пустота	Нет

Если будет использован запрос <http://localhost:8080/WebApp2/>, то появится сообщение:

Ошибка: Необходимо ввести имя после URL в формате "?name=Имя".

Порядок выполнения лабораторной работы

1. Откройте Maven проект для реализации пользовательского интерфейса Web-приложения на основе JSP-страниц.
2. Разработайте JSP-страницу, которая формирует HTTP-запрос и обрабатывает его. Страница должна содержать форму, отправляющую данные на сервер, и отображать результат обработки. Обработка на сервере должна осуществляться на языке Java, а вывод информации об ошибочных ситуациях – выполняться с помощью JSP-страниц.
3. Создайте war-архив и поместите его на сервер Apache Tomcat.
4. Запустите Web-сервер и обратитесь к стартовой JSP-странице из браузера, проверьте правильность отображения исходной формы и результатов работы JSP-страниц.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Скриншоты, иллюстрирующие работу JSP-страниц.
2. Распечатку файлов с JSP-страницами.
3. Текст документации проекта, сгенерированный Javadoc.

Лабораторная работа № 6. АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ ПОЛЬЗОВАТЕЛЕЙ WEB-ПРИЛОЖЕНИЯ

Цель работы: знакомство со способами реализации аутентификации и авторизации пользователей Web-приложения.

Способы аутентификации пользователей

При разработке Web-приложения для обеспечения безопасности часто используется связка из двух процедур – аутентификации и авторизации. Аутентификация позволяет приложению узнать, какой именно пользователь обратился к приложению. Авторизация позволяет определять, какие именно Web-страницы или их категории будут доступны данному конкретному пользователю. Java-платформа и Web-сервер Apache Tomcat предоставляют несколько способов аутентификации пользователей:

- на основе HTTP;
- на основе форм;
- на основе проверки сертификата сервера.

При установке базовой HTTP-аутентификации Web-сервер будет аутентифицировать пользователя, используя его имя и пароль, полученные от Web-клиента. Базовая аутентификация HTTP задействует механизмы, уже встроенные в протокол HTTP. При запросе защищенного ресурса контейнер посылает Web-клиенту ответ 401 HTTP. По этому ответу браузер запрашивает имя пользователя и пароль через стандартное диалоговое окно регистрации и возвращает их в следующем HTTP-запросе контейнеру как часть заголовка этого запроса. При этом браузер передает имена пользователей и пароли по сети Интернет в виде Base64-кодированного, а не зашифрованного, текста.

Аутентификация формой, или, как ее еще называют, аутентификация на основе Cookie-файлов, имеет ряд преимуществ над базовой аутентификацией. Во-первых, имеется возможность определить внешний вид формы регистрации вместо стандартной. Во-вторых, полностью контролируются за-

просы клиента, поскольку инструкции проверки сеанса пользователя находятся в самом сценарии, а не в конфигурационном файле Web-сервера. Аутентификация по форме обычно использует метод HTTP POST и пересылает данные по сети без шифрования.

При аутентификации по форме должна быть разработана JSP-страница, являющаяся по сути HTML-формой и запрашивающая у клиента имя пользователя и пароль. Данные формы передаются сервлету, который проводит аутентификацию, обращаясь к файлу или базе данных, где хранится пароль в зашифрованном виде. Если введенные данные верны, то приложение становится доступным для клиента.

Полностью управляемый процесс аутентификации пользователя легко реализовать с помощью сеансов. Для этого в защищаемые JSP-страницы вставляется блок кода, который проверяет переменные сеанса и правильность аутентификации, перед тем как продолжить процесс. Если пользователь авторизован для просмотра ресурса, который он запрашивал, контейнер обслуживает запрос для защищенного ресурса.

Аутентификация, основанная на проверке сертификата сервера, более безопасна, чем базовая или основанная на форме. Она использует протокол SSL (Secure Sockets Layer), в котором передаваемые данные шифруются, а сервер и (необязательно) клиент аутентифицируют друг друга, используя сертификаты с открытым ключом. Для аутентификации сервера, т. е. для проверки, что сервер, с которым контактирует клиент и передает ему конфиденциальные данные, тот, за который он себя выдает, требуется сертификат ключа шифрования. Сертификат, подписанный доверенным центром авторизации, дает такую гарантию. При более низких требованиях к безопасности можно использовать собственные ключи шифрования и самоподписанные сертификаты, которые самостоятельно создаются владельцами Web-сервера.

С помощью метода `request.getAuthType()` можно определить имя схемы аутентификации, используемой для защиты сервлета.

Включение базовой аутентификации в Web-приложение

Базовая аутентификация включается в Web-приложение настройкой двух файлов сервера Apache Tomcat: `/conf/tomcat-users.xml` и `/WEB-INF/web.xml` проекта. В первом файле определяются пользователи, пароли и роли в виде следующих строк:

```
<role rolename="роль"/>
```

```
<user username="пользователь" password="пароль" roles="роли"/>
```

К Web-приложению могут иметь доступ несколько пользователей с разными ролями. Каждому пользователю можно назначить одну или несколько ролей. Например:

```
<role rolename=" manager "/>
```

```
<role rolename=" admin "/>
```

```
<role rolename=" tomcat "/>
```

```
<user username="admin" password="admin" roles="manager, admin " />
```

```
<user username="myname" password="mypassword" roles="tomcat" />
```

Во втором файле (web.xml), находящемся в папке проекта, указывается способ аутентификации, защищаемые ресурсы, имена ролей, которые позволяют обращаться к этим ресурсам. Например, для разрешения запроса к JSP-странице BookList с помощью методов GET и POST пользователю admin с ролью manager нужно включить в файл /WEB-INF/web.xml следующие строчки:

```
<!-- Описание защищаемых ресурсов -->
```

```
<security-constraint>
```

```
<web-resource-collection>
```

```
<web-resource-name> Список книг </web-resource-name>
```

```
<url-pattern> /BookList.jsp </url-pattern>
```

```
<http-method> GET </http-method>
```

```
<http-method> POST </http-method>
```

```
</web-resource-collection>
```

```
<auth-constraint>
```

```
<role-name> manager </role-name>
```

```
<role-name> tomcat </role-name>
```

```
</auth-constraint>
```

```
</security-constraint>
```

```
<!-- Определение вида аутентификации -->
```

```
<login-config>
```

```
<auth-method>BASIC</auth-method>
```

```
<realm-name> Write Book List </realm-name>
```

```
</login-config>
```

Тег <realm-name> при описании вида аутентификации определяет ее контекст, а значение тега отображается в диалоговом окне и подсказывает пользователю, какой логин\пароль следует использовать в данной ситуации.

После настройки базовой аутентификации при вызове <http://localhost:8080/WebApp2?name=Петров П.П.> в браузере появится следующее окно (рис. 6.1).

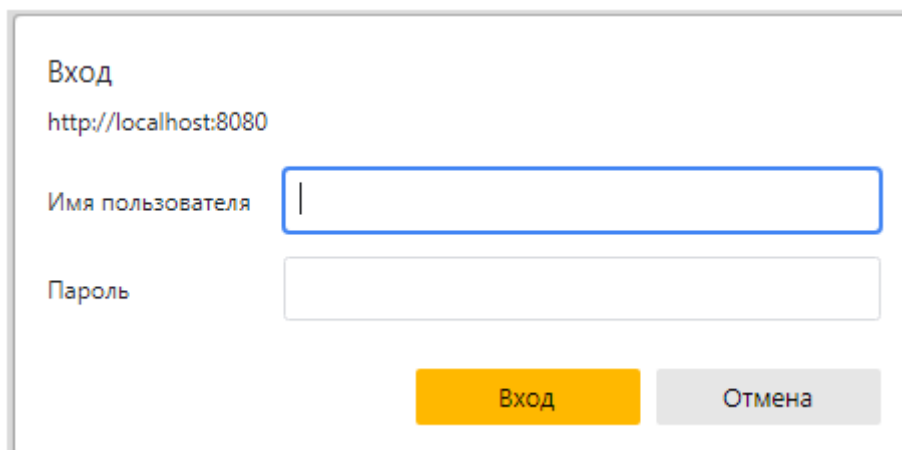


Рис. 6.1. Окно аутентификации

В этом окне надо ввести имя пользователя и пароль, которые соответствуют ролям, прописанным в разделе защищаемых ресурсов (например, `admin`, `admin`). Если имя пользователя недействительно или пароль неверный, то браузер повторно запросит аутентификацию. В случае нажатия кнопки «Отмена» сервер передаст в браузер код статуса HTTP 401, который указывает, что запрос был отклонен. В таких случаях приложения, в которых используется аутентификация по форме, перенаправляют запрос на страницу ошибки, и пользователи продолжают работу, даже если не проходят авторизацию.

Настройка SSL-протокола

Чтобы защитить пользовательские данные, передаваемые посредством HTTP, обычно используют протокол SSL. Для применения SSL нужно создать хранилище для ключа и сам ключ, с помощью которого будут шифроваться передаваемые по HTTP данные, и настроить Web-сервер. Хранилище и ключ создаются с помощью утилиты `keytool`, находящейся в каталоге `/bin` JDK. Команда ее запуска имеет вид

```
keytool -genkey -alias test -keystore mystore -validity 365 -keyalg RSA -keysize 1024
```

В этой команде должны быть заданы следующие параметры (элементы, выделенные жирным курсивом, представляют значения параметров):

- genkey (генерация ключа шифрования);
- alias (имя сертификата ключа, например *test*);
- keystore (имя файла хранилища, например *mystore*);
- validity (срок действия сертификата в днях, например 365 дней, по умолчанию 180 дней);

-keyalg (алгоритм формирования ключа, например *RSA*, по умолчанию *DSA*);

-keysize (размер ключа в битах, например 1024, может отсутствовать).

После запуска утилита запрашивает следующую информацию:

Enter keystore password: (пароль для хранилища не менее 6 символов с повторным вводом и без отображения вводимых символов);

What is your first and last name? (свое имя и фамилию или доменное имя сервера);

What is the name of your organizational unit? (название подразделения организации);

What is the name of your organization? (название организации);

What is the name of your City or Locality? (название города);

What is the name of your State or Province? (название области);

What is the two-letter country code for this unit? (двухсимвольное обозначение страны – ru);

Is CN=firstname lastname, OU=organizationalunit, O=organization, L=city, ST=state, C=ru correct? (подтверждение указанной информации – yes или no);

Enter key password for (RETURN if same as keystore password): (пароль для ключа не вводить или должен совпадать с паролем хранилища).

Проверить наличие информации в хранилище можно с помощью команды: `keytool -list -keystore имя хранилища`

Созданный утилитой `keytool` ключ хранится в файле, имя которого было задано в команде, и этот файл надо положить в папку **conf** корневой директории Web-сервера Apache Tomcat. Затем в файле `server.xml`, расположенном в этой папке, раскомментировать элемент `<Connector port="8443"`, заменить в нем значение атрибута `protocol` на `org.apache.coyote.http11.Http11Protocol` и добавить в него 3 атрибута `keystoreFile`, `keystorePass` и `useBodyEncodingForURI`. Значением атрибута `keystoreFile` должно быть имя хранилища, атрибута `keystorePass` – пароль, который был указан при создании хранилища, а атрибута `useBodyEncodingForURI` – значение "true", которое позволит использовать в параметрах запроса русские буквы. После проделанных операций необходимо сохранить файл, в котором отредактированный элемент будет иметь следующий вид:

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
    SSLEnabled="true"
```

```
maxThreads="150" scheme="https" secure="true"  
clientAuth="false" sslProtocol="TLS"  
keystoreFile="conf/mystore"  
keystorePass="123456"  
useBodyEncodingForURI="true">  
</Connector>
```

Для запуска JSP-страницы по защищенному протоколу необходимо обратиться к Web-серверу по адресу:

<https://localhost:8443/WebApp2?name=Петров П.П.>

Если сертификат выдан не доверенной организацией, браузеры выводят предупреждение, изображенное на рис. 6.1.

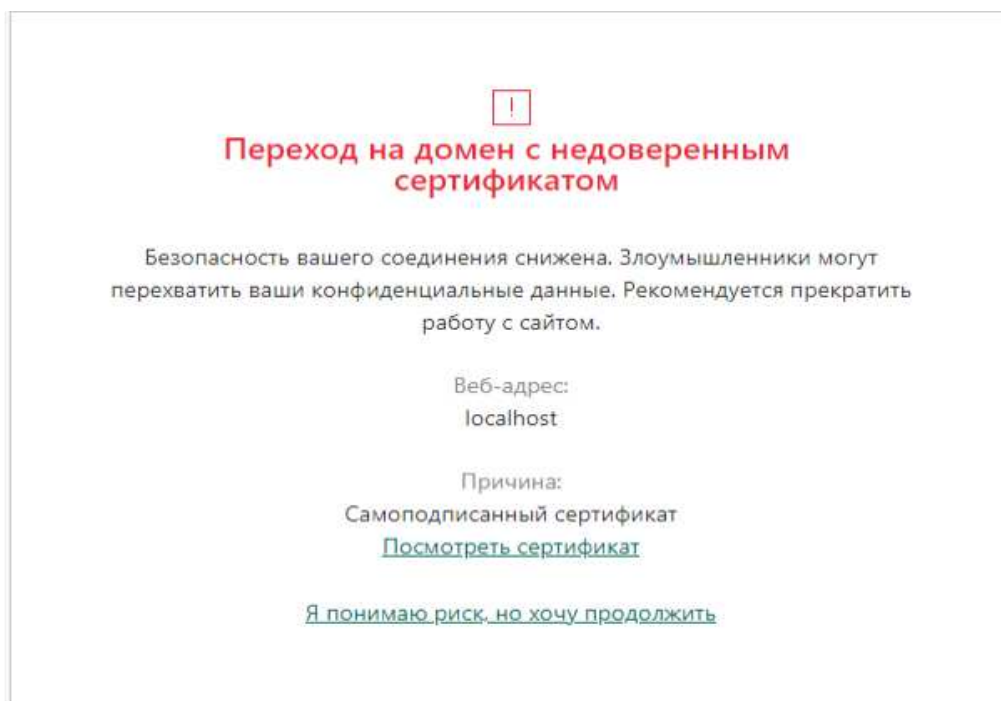


Рис. 6.1. Окно предупреждения

Для продолжения работы надо проигнорировать это предупреждение, тем самым сообщая браузеру, что сервер является безопасным.

Порядок выполнения лабораторной работы

1. Создайте Maven проект с Web-приложением, аналогичным лабораторной работе № 5.
2. Настройте Web-сервер на базовую аутентификацию.
3. Запустите приложение с указанием имени пользователя и пароля.
4. Настройте Web-сервер на протокол SSL и запустите приложение, используя защищенный порт.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Распечатку файлов tomcat-users.xml, web.xml и server.xml.
2. Скриншот, иллюстрирующий создание хранилища сертификата.
3. Скриншот работы приложения.

Лабораторная работа № 7. ОРГАНИЗАЦИЯ ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ЗАПРОСАМИ ПОЛЬЗОВАТЕЛЕЙ

Цель работы: знакомство с методами передачи информации между соединениями, открываемыми в рамках одного сеанса работы пользователя.

Передача данных через файл Cookie

Cookie («куки») – это небольшой блок текстовой информации, которую Web-сервер передает браузеру. Браузер возвращает информацию обратно на сервер как часть заголовка HTTP, когда клиент повторно заходит на тот же Web-ресурс. Одни значения Cookie могут храниться только в течение одной сессии и удаляются после закрытия браузера. Другие, установленные на некоторый период времени, записываются в специальный файл и хранятся в компьютере пользователя. В одном файле Web-сервер может хранить несколько объектов Cookie, каждый из которых размером не более 4 Кбайт. Клиент может запретить браузеру прием файлов Cookie настройкой уровня конфиденциальности. В этом случае браузер будет оповещать об отключении этой функции при попытке установки файлов Cookie.

В Java Cookie являются объектом класса `javax.servlet.http.Cookie`. Чтобы послать Cookie клиенту, необходимо создать объект класса Cookie, указав конструктору его имя в качестве первого параметра и его значение – в качестве второго (оба параметра представляют собой строки). Например:

```
Cookie ck = new Cookie("username", "student");
```

После создания объекта Cookie ему можно задать следующие атрибуты:

- домен, для которого значение Cookie обрабатывается (если атрибут опущен, то по умолчанию используется доменное имя сервера, на котором было задано значение Cookie);
- путь, устанавливающий подмножество URL, для которых предназначен Cookie (если атрибут не указан, то значение Cookie возвращается только для URL, который устанавливал значение Cookie);

- время жизни Cookie в секундах от момента первой отправки клиенту (если атрибут не указан, то Cookie существует только до момента закрытия браузера);
- способ передачи Cookie – либо по протоколу HTTP, либо по протоколу HTTPS (если атрибут не задан, то передача осуществляется по протоколу HTTP).

Вставка Cookie в заголовок HTTP-ответа происходит с помощью метода `addCookie`. Например:

```
response.addCookie(ck);
```

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

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

К нему можно применить метод `getName` и по совпадающему имени получить с помощью метода `getValue()` содержимое блока Cookie.

Передача данных через сессионный объект

Важнейшая задача, которую приходится решать при создании Web-приложений, это отслеживание запросов пользователя в течение всего сеанса его работы. Сеанс (session) – это последовательность запросов, отправляемых конкретным клиентом серверу, и ответов, полученных от Web-сервера. Каждый запрос, получаемый Web-сервером по протоколу HTTP, является независимым элементом данных, не связанным с ранее поступившими запросами. В связи с этим Web-сервер не может установить, от какого пользователя поступил запрос. Технология Java Servlets позволяет ассоциировать каждый запрос с сеансом и предоставляет объект `HttpSession` для хранения данных, относящихся к этому сеансу. Можно привязать пользовательские данные к объекту `HttpSession` и получить возможность доступа к этим данным на дальнейших этапах работы пользователя. Операции привязки и доступа могут быть выполнены при помощи классов сервлета, а также неявной переменной `session` в JSP-страницах.

Чтобы найти объект `HttpSession`, который работает с данным запросом, вызывается метод `getSession()` интерфейса `HttpServletRequest`. Если метод возвращает `null`, то такого объекта нет. Проверку наличия сессионного объекта можно совместить с его созданием. Для этого необходимо в качестве аргумента метода `getSession` указать значение `true`:

```
HttpSession session = request.getSession(true);
```

В этом случае сервлет проверяет наличие сессионного объекта, привязанного к данному клиенту. В случае успеха метод `getSession` возвращает ссылку на этот объект. В противном случае метод создает новый сессионный объект. Для правильной привязки сессионного объекта к запросу надо вызвать метод `getSession`, прежде чем будет запущен выходной поток ответа.

Сессионный объект хранится на сервере персонально для каждого клиента. Сервер различает сессии с помощью идентификатора, который передается пользователю. Идентификатор сессии автоматически сохраняется в браузере пользователя в виде Cookie, а если браузер запрещает Cookie, то Web-сервер автоматически помещает данный идентификатор в переменную JSESSIONID каждой ссылки на выдаваемых клиенту HTML-страницах. Это значит, что при обновлении страницы браузер сам отправит на сервер идентификатор сессии либо из Cookie, либо из адресной строки ссылки независимо от действий пользователя. Как только Web-сервер получает от клиента определенный идентификатор сессии, то он передает сценарию на той странице, на которую зашел клиент, ссылку на сессионный объект, созданный для этого клиента.

Сессионный объект обеспечивает средства для хранения и доступа к данным пользователя на протяжении всего периода работы с приложением. Он имеет метод `setAttribute(String, Object)` для добавления данных и метод `getAttribute(String)` – для их извлечения. Подробное описание методов класса `HttpSession` можно найти по адресу <http://bourabai.kz/xml/app1.htm>.

Следует иметь в виду, что отследить момент, когда пользователь перестает работать с приложением, не всегда возможно. Поэтому в настройках `web.xml`-файла сервера устанавливается некоторое предельное время существования объекта `HttpSession` после получения последнего запроса. Это время задается следующим образом:

```
<session-config>
  <session-timeout>ВРЕМЯ в минутах</session-timeout>
</session-config>
```

Если `session-timeout` не настроен, то спецификация Servlet требует, чтобы использовался тайм-аут, выбранный поставщиком контейнера (для Tomcat это 30 мин). Время простоя также может быть настроено с помощью метода `setMaxInactiveInterval` класса `HttpSession`. В отличие от элемента `session-timeout` этот метод принимает значение времени в секундах.

Пример использования файла Cookie и сессионных объектов

В качестве примера рассмотрим приложение WebApp3, которое отображает список прочитанных книг указанного автора. Это приложение с помощью ссылки <http://localhost:8080/WebApp3/> вызывает JSP-страницу BookList1.jsp, которая запрашивает имя автора. Если пользователь уже вводил имя автора на этой странице, то по умолчанию в поле ввода подставляется последнее имя автора, сохраненное в Cookie. Содержимое файла Cookie можно посмотреть в настройках браузера. Для Яндекс нажимаем в верхнем правом углу на кнопку с тремя линиями и открываем «Настройки». Затем последовательно переходим по пунктам меню «Сайты»→«Расширенные настройки сайтов»→«Cookie-файлы»→«Cookie-файлы и данные сайтов» → «localhost» (Рис. 7.1).

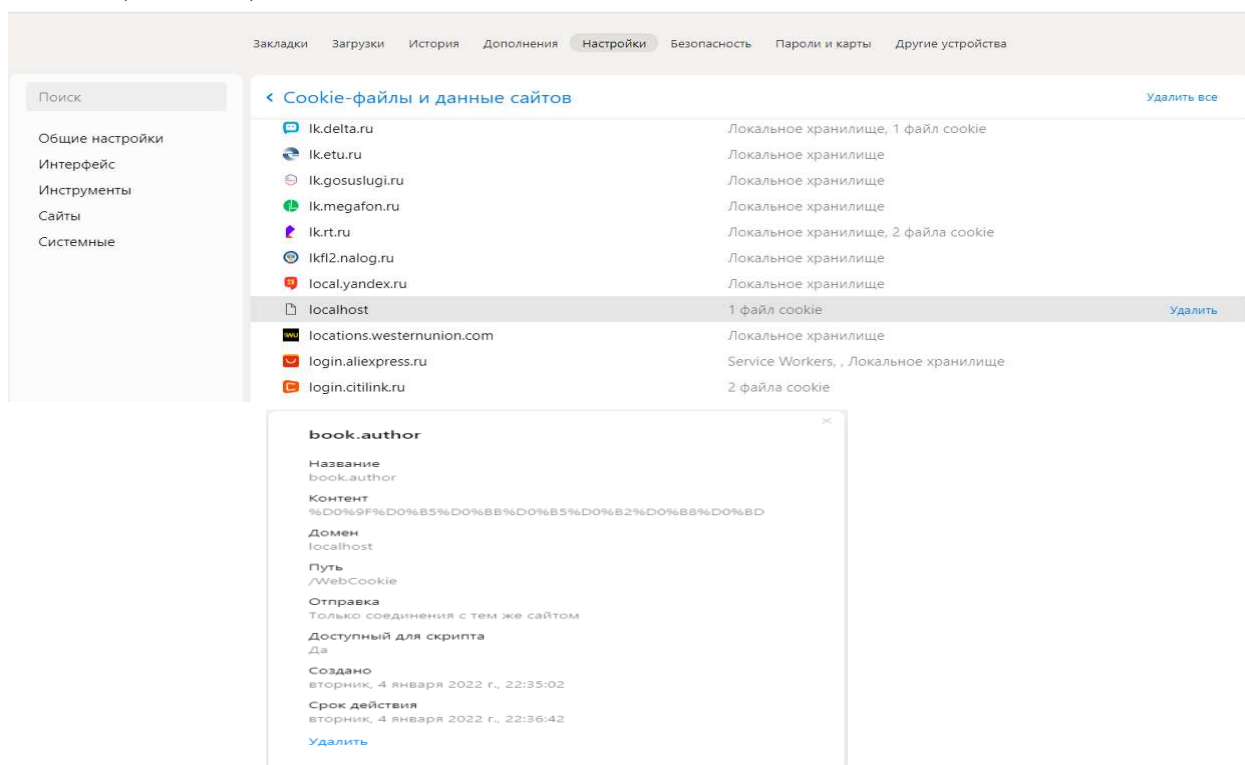


Рис. 7.1 Окно отображения значений Cookie

После нажатия кнопки «Ввод» эта страница вызывает сервлет AuthorProcessor.java. Сервлет осуществляет проверку введенного значения и в случае его отсутствия выводит сообщение «Не задано имя автора». Затем проверяется, есть ли указанный автор в списке авторов. Если автора нет в списке, выводится сообщение «Автор с указанной фамилией не найден». При правильном задании фамилии автора выполняется его запись в сессионный объект и в файл Cookie. Сервлет заканчивает свою работу вызовом JSP-

страницы AskAuthorName.jsp. Эта страница использует значение сессионного объекта для вывода перечня книг указанного автора.

Ниже приведены исходные тексты компонентов приложения.

BookList1.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Введите имя автора</title>
</head>
<body>
    <form METHOD=GET action="AuthorProcessor">
        Введите ФИО автора книги <br>
        <INPUT TYPE=TEXT NAME="author"
            <%
                // Выбор всех Cookie
                Cookie [] c = request.getCookies();
                if(c != null)
                    for(int i = 0; i < c.length; i++)
                        if("book.author".equals(c[i].getName())) {
                            // Запись значения в поле ввода, если найден Cookie
                            out.print(" value=" + java.net.URLDecoder.decode(c[i].getValue(), "UTF-8") + " ");
                            break;
                        }
            %>
        > <br>
        <INPUT TYPE=SUBMIT value="Ввод">
    </form>
</body>
</html>
```

AuthorProcessor.java

```
package edu.etu.web7;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Cookie;
import java.net.URLEncoder;

/**
 * Servlet implementation class AuthorProcessor
 */
public class AuthorProcessor extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```

    // Авторы
    private static final String [] authors = {"Булгаков", "Пелевин"};
    public AuthorProcessor() {
        super();
    }
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        request.setCharacterEncoding("utf-8");
        // Получение параметра из строки запроса
        String parameter = request.getParameter("author");
        System.out.println("Параметр "+parameter);
        if(parameter == "") {
            // Сообщение об ошибке, если сервлет вызван без параметра
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Не задано имя
автора");
            return;
        }
        // Проверка, что такой автор есть
        boolean haveBooks = false;
        for(int i = 0; i < authors.length; i++)
            if(parameter.equals(authors[i])){
                haveBooks = true;
                break;
            }
        if(!haveBooks) { // Сообщение об ошибке, если автор не найден
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Автор с фамилией " +
parameter + " не найден");
            return;
        }
        // Сохранение автора в сессии
        request.getSession().setAttribute("author", parameter);
        // Сохранение автора в Cookie
        Cookie c = new Cookie("book.author", URLEncoder.encode(parameter, "UTF-8"));
        // Установка времени жизни Cookie в секундах
        c.setMaxAge(100);
        response.addCookie(c);
        // Перенаправление на страницу книг
        response.sendRedirect(response.encodeRedirectURL(request.getContextPath() +
"/AskAutorName.jsp"));
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        processRequest(request, response);
    }

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

```

}

AskAutorName.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Отфильтрованный список книг</title>
</head>
<body>
    <h1>Список книг</h1>
    <table border='1'>
        <tr>
            <td><b>Автор книги</b></td>
            <td><b>Название книги</b></td>
            <td><b>Прочитал</b></td>
        </tr>
        <% if("Булгаков".equals((String)session.getAttribute("author"))) {%>
        <tr>
            <td>Булгаков</td>
            <td>Мастер и Маргарита</td>
            <td>Да</td>
        </tr>
        <% }
        if("Пелевин".equals((String)session.getAttribute("author")))
        {%>
        <tr>
            <td>Пелевин</td>
            <td>Чапаев и пустота</td>
            <td>Нет</td>
        </tr>
        <% } %>
    </table>
</body>
</html>
```

Порядок выполнения лабораторной работы

1. Создайте Maven проект и приложение, в котором JSP-страница запрашивает имя пользователя и цвет страницы вывода информации с сохранением этих параметров в Cookie. Эта же страница должна в переменных сессии подсчитывать количество обращений пользователей и дату последнего обращения, а затем передать управление другой JSP-странице, которая будет отображать содержимое Cookie и переменных

сессии в заданном цвете.

2. Снимите несколько скриншотов окна настроек браузера, в котором отображаются различные значения Cookie, а также скриншоты результатов работы приложения при различных входных параметрах.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Тексты JSP-страниц с комментариями.
2. Скриншоты, иллюстрирующие работу приложения.

Лабораторная работа № 8. РАЗРАБОТКА Web-ПРИЛОЖЕНИЯ С ИСПЛЬЗОВАНИЕМ GWT

Цель работы: знакомство с процессом создания GWT-приложения в среде Eclipse и Maven.

Технология GWT

Технология Google Web Toolkit (GWT) – это набор средств и интерфейсов для разработки Web-приложений. Ее особенностью является то, что вся разработка серверной и клиентской частей приложения ведется на языке Java. При этом весь клиентский код, выполняющийся в среде Web-браузера, преобразуется компилятором из Java в JavaScript и HTML, что позволяет разработчикам использовать для клиентской части объектно-ориентированную модель программирования с возможностью прямого доступа к серверным данным с помощью сервлет-контейнера. Для взаимодействия клиентской и серверных частей используется удаленный вызов процедур RPC (Remote Procedure Call). Данный функционал освобождает от проблем с поддержкой JavaScript-кода в различных браузерах и позволяет абстрагироваться от HTTP-протокола и HTML DOM-модели.

Создание шаблона проекта GWT-приложения

Шаблон GWT-проекта может быть построен на основе фреймворка GWT SDK, который содержит библиотеки и компилятор для написания GWT-приложения. Для установки GWT SDK необходимо выбрать пункт меню Help->Eclipse Marketplace и в строке поиска открывшемся окне (рис. 8.1) набрать GWT и выполнить установку GWT Eclipse Plugin 3.0.0.

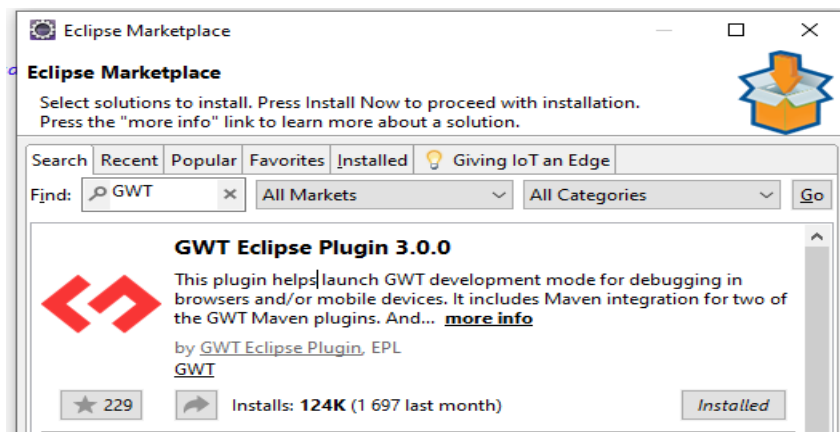


Рис. 8.1 Установка плагина GWT Eclipse Plugin 3.0.0

При успешной установке на кнопке инсталляции должна появиться надпись *Installed*. Далее создается и настраивается Maven проект с архетипом `gwt-maven-plugin` версия 2.9.0. Надо выбрать пункты меню `File`→`NEW`→`Project` и в появившемся окне раскрыть секцию `Maven`, в которой выделить пункт `Maven Project`. Затем нажать кнопку `Next` в текущем и следующем окне. Появится окно выбора архетипа (рис.8.2). В этом окне надо выбрать архетип `gwt-maven-plugin` версия 2.9.0 и нажать кнопку `Next`.

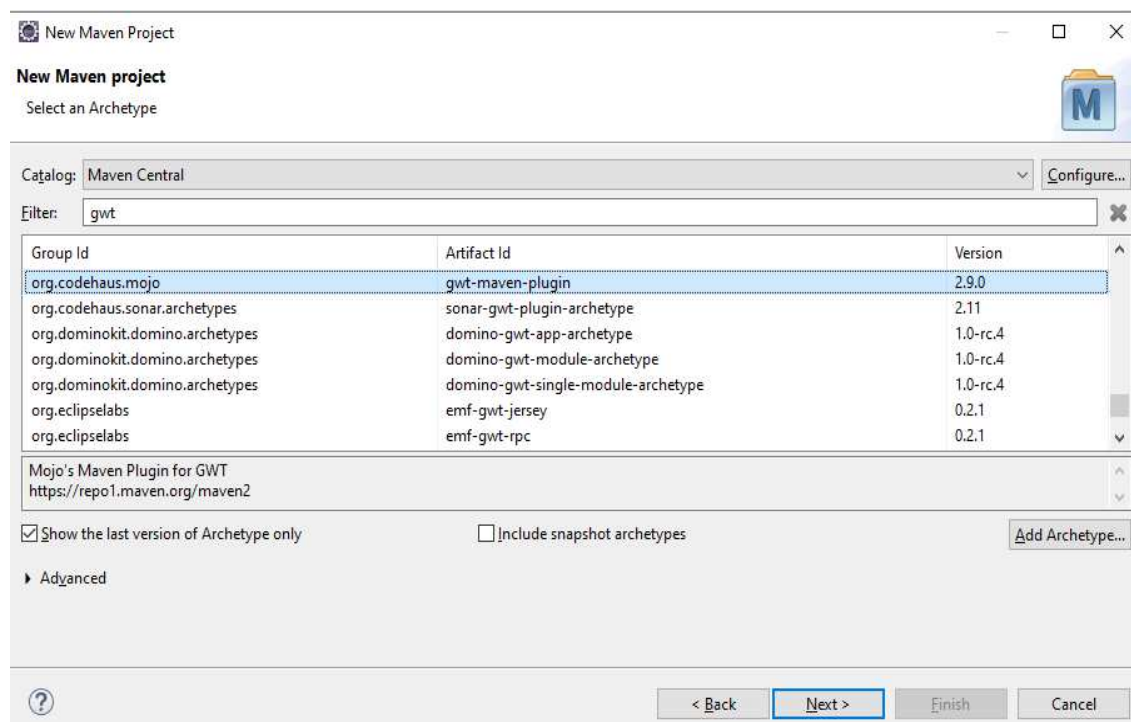


Рис. 8.2 Окно выбора архетипа

Во вновь открывшемся окне (рис. 8.3) заполнить поля: `Group Id` (имя организации, например, ETU), `Artifact Id` (имя проекта, например, WebGWT), `Value` (имя модуля клиентской части, например, BookModule) и нажать кноп-

ку Finish. Структура проекта, созданная плагином Maven GWT представлена на рис. 8.4.

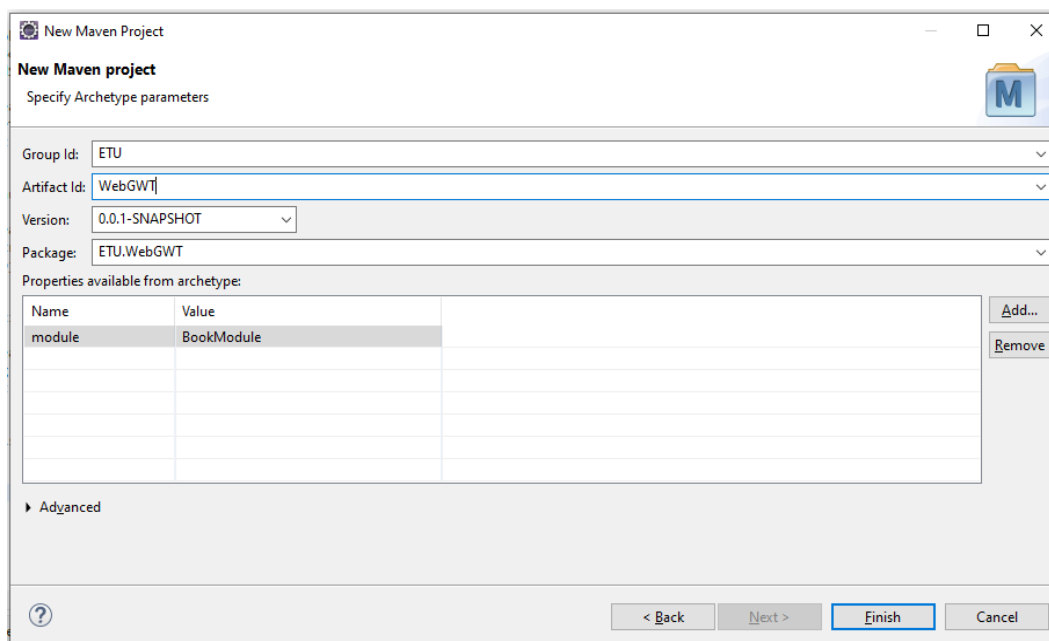


Рис. 8.3 Окно спецификации параметров GWT-проекта

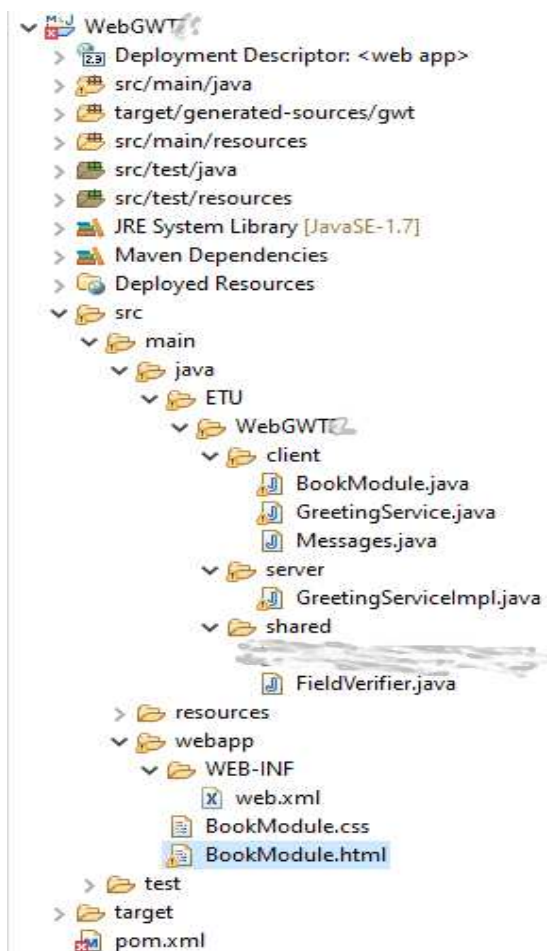


Рис. 8.4 Структура проекта GWT

Исходная папка приложения `src/main/java` содержит файлы Java. Такие файлы разделены на 3 категории: клиентские, серверные и общие. Клиентские файлы будут перенесены с Java на JavaScript и будут выполняться в браузере пользователя. В клиентской части примера размещается главный класс приложения `BookModule.java`, интерфейс Web-сервиса (`GreetingService.java`), поставляющий данные клиенту, и вспомогательный интерфейс (`Messages.java`) для представления сообщений. Созданный класс наследует интерфейс `EntryPoint`, в котором определен метод `onModuleLoad()`. GWT автоматически вызывает данный метод при загрузке HTML-страницы, ссылающейся на данный модуль.

Файлы сервера будут скомпилированы в байт-код и будут выполняться на сервере как обычные файлы Java. Созданный средой Eclipse класс `GreetingServiceImpl.java` расширяет свойства класса `com.google.gwt.user.server.rpc.RemoteServiceServlet` и реализует интерфейс Web-сервиса. Класс `RemoteServiceServlet` обеспечивает десериализацию входящих клиентских запросов и сериализацию ответов клиенту.

Общие файлы будут совместно использоваться на стороне клиента и на стороне сервера с помощью вызова процедуры RPC. Сгенерированный средой Eclipse класс `FieldVerifier.java` обеспечивает проверку введенных пользователем данных.

Папка `src/main/webapp` содержит файл `BookModule.html`, используемый для запуска веб-приложения, файл CSS-стилей и файл дескриптора развертывания `web.xml`.

Для созданного проекта надо разрешить включить модуль точки входа. Для этого надо открыть настройки проекта `preferences->Java Build Path`, выбрать каталог `src/main/sources`, выделить строку `Excluded`, а затем нажать на кнопку `Remove` (рис. 8.5). Вместо `Excluded **` должна появиться надпись `Excluded (None)`. Сохранение настройки происходит после нажатия кнопки `Apply and Close`.

Следующим шагом скачиваем архив GWT-2.9.0 с <http://www.gwtproject.org/download.html>. После того как выполнилось скачивание и файл был разархивирован, нажимаем правой кнопкой мыши по проекту, выбираем `GWT->Settings` после чего появляется окно (рис. 8.6).

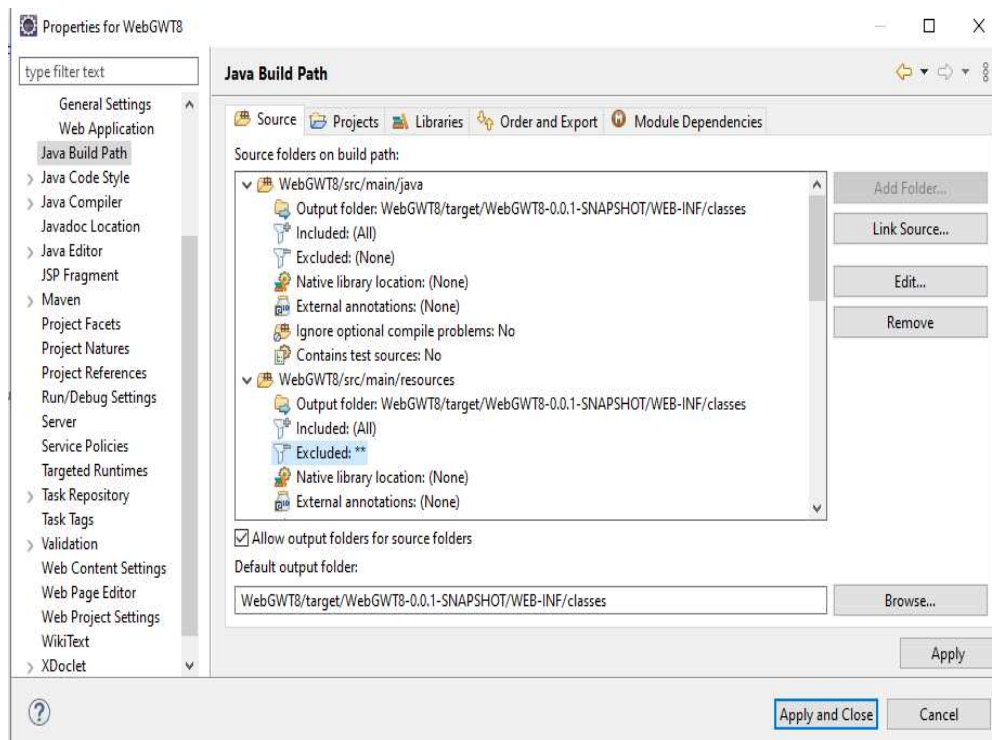


Рис. 8.5 Окно настройки проекта

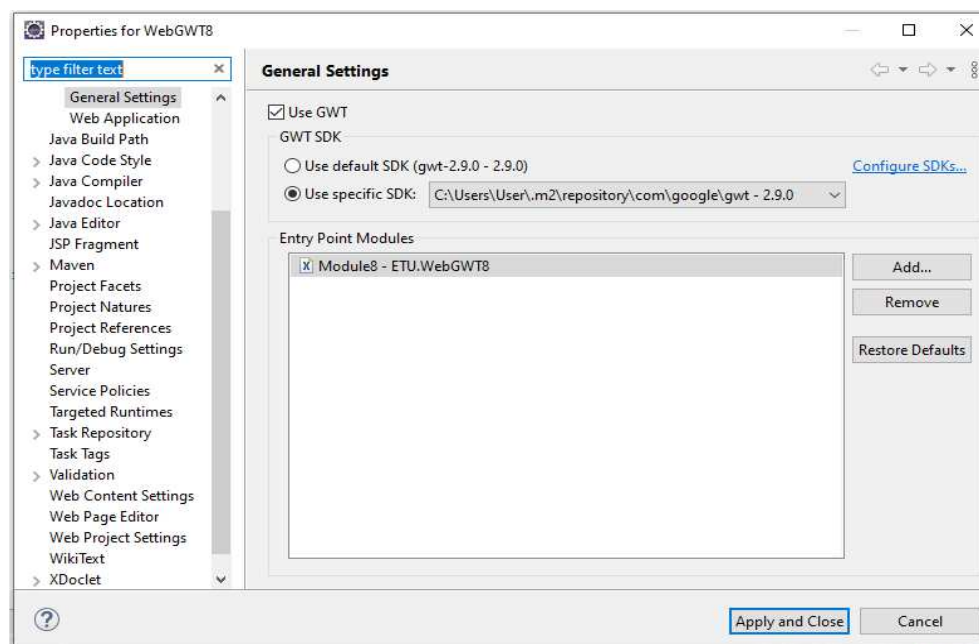


Рис. 8.6 Окно установок GWT SDK

В этом окне переходим по ссылке [Configure SDKs...](#), откроется окно (рис. 8.7) подключенных библиотек, нажимаем на кнопку add и указываем путь к распакованному архиву GWT 2.9.0. Проверяем, что версия 2.9.0 выделена в настройках и, если всё правильно, нажимаем Apply and Close.

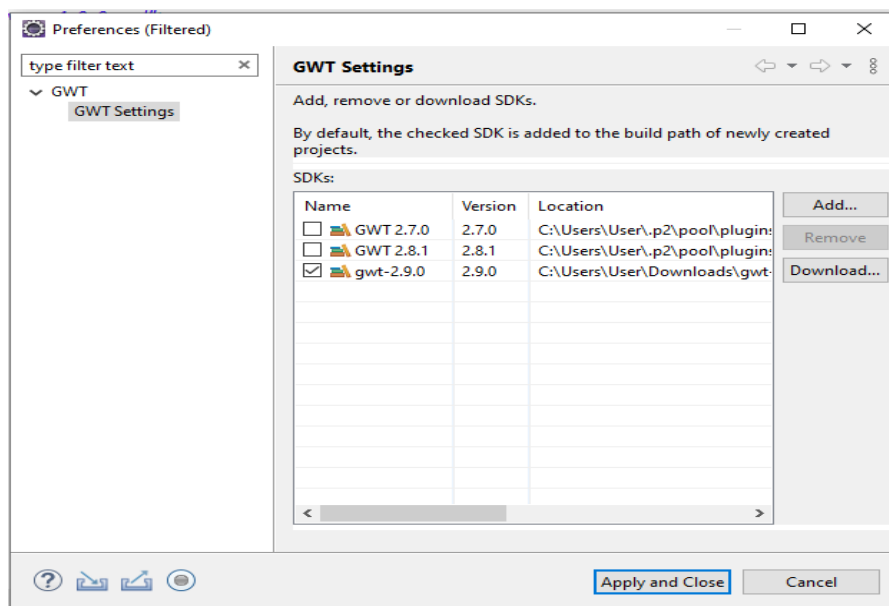


Рис. 8.7 Окно установок GWT 2.9.0.

Для сборки проекта нажимаем правой кнопкой мыши и выбираем Run As->GWT Development Mode With Jetty. В сплывающем окне нажимаем Proceed. После этого во вкладке Development Mode появится URL приложения (рис. 8.8).

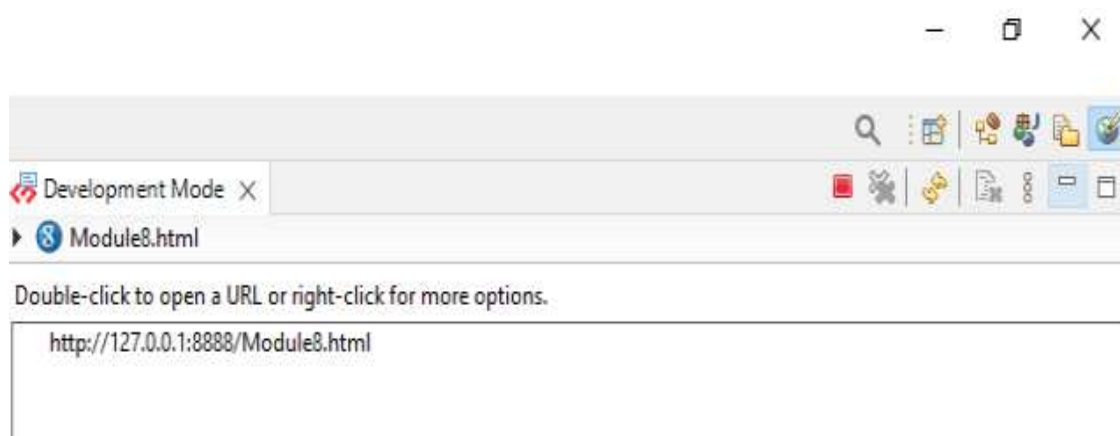


Рис. 8.8 Окно запуска приложения

В браузере приложение должно стать доступным по адресу <http://127.0.0.1:8888/Имя модуля.html>. Запустить приложение можно по двойному щелчку мыши на строке URL, после чего откроется окно браузера и появится сообщение, представленное на рис. 8.9. В данном GWT-приложении определены объекты Button и Label, а также обработчик событий, который вызывается при нажатии кнопки «Send». Этот обработчик посылает имя пользователя на сервер и в ответ получает от него сообщение (рис. 8.10), которое выводится клиентом в диалоговом окне.

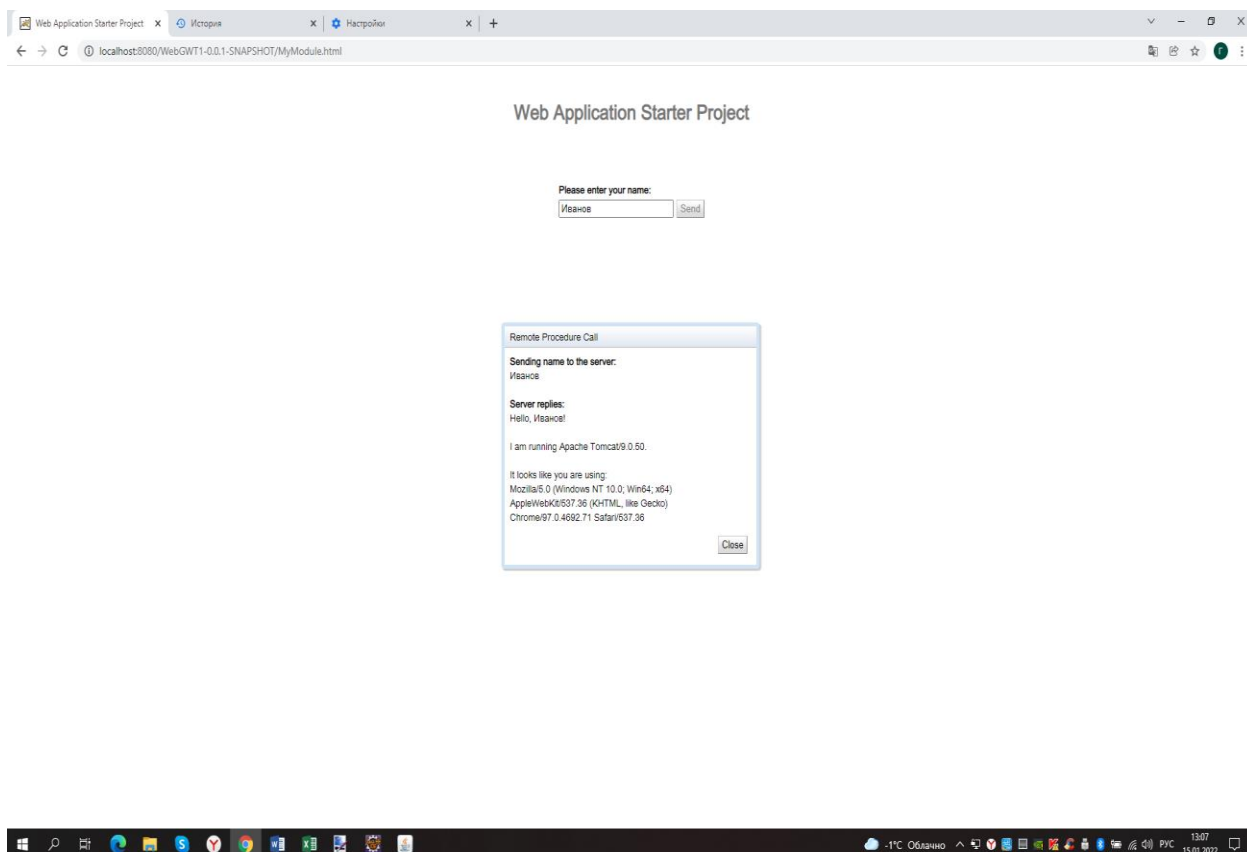


Рис. 8.10 Сообщение сервера клиенту

Данный шаблон проекта носит универсальный характер и может быть использован в качестве начального каркаса для разработки своего GWT-приложения.

Разработка клиентской части GWT-приложения

В качестве примера рассмотрим разработку GWT-приложения, отображающего список книг, прочитанных или непрочитанных читателем. Эту программу можно построить путем изменения сгенерированных при создании шаблона проекта файлов. В клиентской части надо модернизировать следующие файлы: BookModule.html, BookModule.java и GreetingService.java. Прежде всего надо решить вопрос, что будет храниться на сервере и каким образом клиент будет взаимодействовать с сервером. На сервере предполагается хранить список читателей и перечень книг, с которыми он должен ознакомиться. Клиент должен сначала получить список читателей, затем выбрать одного из читателей и получить для него список книг. Это взаимодействие описывается в файле GreetingService.java в виде следующего интерфейса:

```
package ETU.WebGWT.client;  
import java.util.List;
```

```

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
import ETU.WebGWT.shared.BookReader;
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
    List<String> getReaderList();
    List<BookReader> getBookReaderList(String readerFIO) throws IllegalArgumentException;
}

```

Аннотация `@RemoteServiceRelativePath("greet")` в интерфейсе указывает путь к сервлету. Методами этого интерфейса являются `getReader` и `getBookReaderList`. Первый метод запрашивает у сервера список читателей, а второй – список его книг. GWT имеет две особенности реализации интерфейса `GreetingService`:

- для обращения к серверу используется механизм вызова удаленных процедур, поэтому он должен наследовать интерфейс `RemoteService`;
- GWT реализует асинхронные интерфейсы для обращения к удаленным сервисам, т. е. не дожидаясь ответа от первого посланного запроса клиент может послать другой или тот же самый запрос на сервер, при этом отсутствуют какие-либо блокировки пользовательского интерфейса.

Таким образом, клиент обращается к методам через асинхронный интерфейс, а GWT преобразует их и передает на сервер вызовы методов удаленного интерфейса, где они выполняются. Maven при выполнении команды `install` автоматически строит описание асинхронного интерфейса и помещает его в файл `/WebGWT/target/generated-sources/gwt/ETU/WebGWT2/client/GreetingServiceAsync.java`. Описание этого интерфейса выглядит так:

```

package ETU.WebGWT.client;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
public interface GreetingServiceAsync
{
    void getReaderList( AsyncCallback<java.util.List<java.lang.String>> callback );
    void getBookReaderList( java.lang.String readerFIO, AsyncCallback <java.util.List
<ETU.WebGWT2.shared.BookReader>> callback );
    public static final class Util
    {
        private static GreetingServiceAsync instance;
        public static final GreetingServiceAsync getInstance()
        {
            if ( instance == null )

```

```

        {
            instance = (GreetingServiceAsync) GWT.create( GreetingService.class );
        }
        return instance;
    }
    private Util()
    {
        // Utility class should not be instantiated
    }
}
}

```

Поскольку описание этого интерфейса отсутствует в клиентском модуле, то редактор Eclipse может выдавать следующее сообщение «GreetingServiceAsync cannot be resolved to a type». Оно пропадает после выполнения команды Maven install. В файле GreetingService.java также добавлена ссылка, где содержится описание класса BookReader. Этот класс хранится в файле BookReader.java и определяет состав полей книги и операции доступа к ним.

```

}

```

В файле BookModule.java надо переопределить метод onModuleLoad(), в котором на языке Java описать интерфейс пользователя:

```

package ETU.WebGWT2.client;
import ETU.WebGWT2.shared.BookReader;
import java.util.List;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.cellview.client.CellTable;
import com.google.gwt.user.cellview.client.TextColumn;
import com.google.gwt.user.cellview.client.HasKeyboardSelectionPolicy.KeyboardSelectionPolicy;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DialogBox;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.ListBox;
import com.google.gwt.user.client.ui.MultiWordSuggestOracle;
import com.google.gwt.user.client.ui.RootPanel;

```

```

import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class BookModule implements EntryPoint {
    /**
     * The message displayed to the user when the server cannot be reached or
     * returns an error.
     */
    private static final String SERVER_ERROR = "An error occurred while "
        + "attempting to contact the server. Please check your network "
        + "connection and try again.";

    /**
     * Create a remote service proxy to talk to the server-side Greeting service.
     */
    private final GreetingServiceAsync greetingService = GWT.create(GreetingService.class);
    private final Messages messages = GWT.create(Messages.class);
    private static final String SRV_ERR = "Ошибка сервера! ";
    private static final String SRV_ERR_GET_READER_LIST = "Невозможно получить спи-
сок читателей.";
    private static final String SRV_ERR_GET_BOOK_LIST = "Невозможно получить
список книг читателя.";
    private static final String GET_READER_LIST_BTN = "Получить список книг.";
    private static final String CLOSE_BTN = "Закрыть";
    private static final String BOOKS_WND_TITLE = "Список книг читателя ";

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
        final ListBox readerListBox = new ListBox(false);
        final MultiWordSuggestOracle oracle = new MultiWordSuggestOracle();
        final Label errorLabel = new Label();
        final Button sendButton = new Button(GET_READER_LIST_BTN);
        sendButton.addStyleName("sendButton");
        RootPanel.get("readerListBoxContainer").add(readerListBox);
        RootPanel.get("errorLabelContainer").add(errorLabel);
        RootPanel.get("sendButtonContainer").add(sendButton);
        readerListBox.setFocus(true);

        greetingService.getReaderList(new AsyncCallback<List<String>>() {
            public void onFailure(Throwable caught) {
                errorLabel.setText (SRV_ERR+SRV_ERR_GET_READER_LIST);
            }
            public void onSuccess(List<String> result) {
                oracle.clear();
                oracle.addAll(result);
                for(String r : result){readerListBox.addItem(r);

```



```

        }
    }
});
final DialogBox dialogBox = new DialogBox();
dialogBox.setText(BOOKS_WND_TITLE);
dialogBox.setAnimationEnabled(true);
final Button closeButton = new Button(CLOSE_BTN);
closeButton.getElement().setId("closeButton");
final HTML serverResponseLabel = new HTML();
VerticalPanel dialogVPanel = new VerticalPanel();
dialogVPanel.addStyleName("dialogVPanel");

final CellTable<BookReader> table = createCellTable();
dialogVPanel.add(table);

dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
dialogVPanel.add(closeButton);
dialogBox.setWidget(dialogVPanel);

closeButton.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        dialogBox.hide();
        sendButton.setEnabled(true);
        sendButton.setFocus(true);
    }
});

class RPCClickHandler implements ClickHandler, KeyUpHandler {

    public void onClick(ClickEvent event) {
        sendReaderFIOToServer();
    }
    public void onKeyUp(KeyUpEvent event) {
        if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
            sendReaderFIOToServer();
        }
    }
    private void sendReaderFIOToServer() {
        errorLabel.setText("");
        final String readerFIO = reader-
ListBox.getValue(readerListBox.getSelectedIndex());
        sendButton.setEnabled(false);
        greetingService.getBookReaderList(readerFIO,
            new AsyncCallback<List<BookReader>>() {
                public void onFailure(Throwable caught) {
                    dialogBox.setText(SRV_ERR);
                    serverResponseLa-
bel.addStyleName("serverResponseLabelError");
                    serverResponseLabel.setHTML(SRV_ERR+SRV_ERR_GET_BOOK_LIST);

```

```

        dialogBox.center();
        closeButton.setFocus(true);
    }

    public void onSuccess(List<BookReader> result) {

        dialogBox.setText(BOOKS_WND_TITLE + readerFIO);
        table.setRowCount(result.size(), true);
        table.setRowData(0, result);
        dialogBox.center();
        closeButton.setFocus(true);
    }
});

    }
}

RPCClickHandler handler = new RPCClickHandler();
sendButton.addClickHandler(handler);
}

private CellTable<BookReader> createCellTable() {
    final CellTable<BookReader> table = new CellTable<BookReader>();
    table.setKeyboardSelectionPolicy(KeyboardSelectionPolicy.ENABLED);

    TextColumn<BookReader> authorColumn = new TextColumn<BookReader>() {
        public String getValue(BookReader object) {
            return object.getAuthor();
        }
    };
    table.addColumn(authorColumn, "Автор книги");

    TextColumn<BookReader> titleColumn = new TextColumn<BookReader>() {
        public String getValue(BookReader object) {
            return object.getTitle();
        }
    };
    table.addColumn(titleColumn, "Название книги");

    TextColumn<BookReader> isReadColumn = new TextColumn<BookReader>() {
        public String getValue(BookReader object) {
            return object.isReader() ? "Да": "Нет";
        }
    };
    table.addColumn(isReadColumn, "Прочитал");
    return table;
}

}

```

В приведенном фрагменте интерфейс пользователя включает в себя выпадающий список с фамилиями авторов книг (элемент

readerListBoxContainer), кнопку получения списка прочитанных книг (элемент sendButtonContainer) и таблицу, отображающую этот список. GWT узнает о том, что класс BookModule является точкой входа в приложение, из файла WebGWT/src/main/resources/ETU/WebGWT/BookModule.gwt.xml, в котором в элементе entry-point прописывается имя запускаемого класса.

В заключении разработки клиентской части необходимо скорректировать описание сгенерированной HTML-страницы, находящейся в файле WebGWT/src/main/webapp/BookModule.html. В этом файле необходимо заменить следующие элементы:

1. Заголовок вкладки приложения `<title> Web Application Starter Project </title>`) – на «Список книг».
2. Надпись на HTML-странице `<h1> Web Application Starter Project</h1>` – на «Сервис для отображения списка книг читателя».
3. Надпись `<td ... Please enter your name </td>` – на «Выберите читателя».
4. Идентификатор компонента `<td id="nameFieldContainer">` – на «readerListBoxContainer» для отображения выпадающего списка с именами читателей. Идентификатор кнопки "sendButtonContainer" остался такой же, как в шаблонном проекте.

Разработка серверной части GWT-приложения

Серверная часть GWT-приложения представлена классом GreetingServiceImpl, который хранится в пакете src.edu.etu.web.server и содержит описание реализации удаленного интерфейса:

```
package ETU.WebGWT.server;
import ETU.WebGWT.client.GreetingService;
import ETU.WebGWT.shared.BookReader;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet implements GreetingService {
    private Map<String, List<BookReader>> db = null;
    public List<String> getReaderList() throws IllegalArgumentException {
        if( db == null ){
```

```

        initDB();
    }
    String[] tmp = new String[db.keySet().size()];
    db.keySet().toArray(tmp);
    return Arrays.asList(tmp);
}
public List<BookReader> getBookReaderList(String readerFIO) throws
IllegalArgumentException {
    if( db == null ){
        initDB();
    }
    if (readerFIO.equals("Иванов И. И.")||readerFIO.equals ("Петров П.П.") ||
readerFIO.equals("Сидоров С.С. ")) return db.get(readerFIO);
    else throw new IllegalArgumentException ("Нет читателя "+readerFIO);
}
private void initDB(){
    db = new HashMap<String, List<BookReader>>();
    List<BookReader> entries1 = new ArrayList<BookReader>();
    entries1.add(new BookReader("Достоевский", "Игрок", true));
    entries1.add(new BookReader("Толстой", "Анна Каренина", true));
    entries1.add(new BookReader("Достоевский", "Идиот", false));
    db.put("Иванов И. И.", entries1);

    List<BookReader> entries2 = new ArrayList<BookReader>();
    entries2.add(new BookReader("Толстой", "Анна Каренина", false));
    entries2.add(new BookReader("Достоевский", "Игрок", true));
    db.put("Петров П.П.", entries2);

    List<BookReader> entries3 = new ArrayList<BookReader>();
    entries3.add(new BookReader("Толстой", "Анна Каренина", false));
    entries3.add(new BookReader("Булгаков", "Белая гвардия", true));
    db.put("Сидоров С.С.", entries3);
}
}

```

Объекты, передаваемые между сервером и клиентом через механизм GWT-RPC, должны:

- прямо или косвенно реализовывать интерфейс `IsSerializable` или `Serializable`;
- иметь сериализуемые члены;
- иметь конструктор по умолчанию.

Эти объекты определяются в отдельном классе (в данном примере это `BookReader`) и размещаются в папке `shared`.

```

package ETU.WebGWT.shared;
import java.io.Serializable;
public class BookReader implements Serializable{

```

```

    private static final long serialVersionUID = 1L;
    private String author;
    private String title;
    private boolean isReader;
    public BookReader(){
    }
    public BookReader(String _author, String _title, boolean _isReader) {
        this.isReader = _isReader;
        this.title = _title;
        this.author = _author;
    }
    public String getAuthor() {
        return author;
    }
    public String getTitle() {
        return title;
    }
    public boolean isReader() {
        return isReader;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}

```

Сериализация – это процесс упаковки содержания объекта. При сериализации объектов формируется *уникальный последовательный номер версии*, который отражает типы полей и сигнатуры методов. Этот номер версии формируется посредством особого алгоритма хеширования, носящего название SHA (Secure Hash Algorithm). Для каждого класса номер версии будет уникальным. Для совместимости различных версий класса это поле должно иметь одно и то же значение. Если в классе есть статическое поле serialVersionUID, то уникальный последовательный номер версии не вычисляется, а используется указанное в статическом поле значение. Таким образом, система сериализации сможет считывать объекты разных версий класса. При отсутствии поля serialVersionUID для сериализуемого класса компилятор будет выдавать предупреждения. Чтобы исключить такое предупреждение передклассом объявляют аннотацию @SuppressWarnings("serial").

Сборка и запуск GWT-приложения

Сборка проекта осуществляется по команде Run As->GWT Development Mode With Jetty. В всплывающем окне нажимаем Proceed. После этого во вкладке Development Mode появится URL приложения <http://127.0.0.1:8888/BookModule.html> Запустить приложение можно по двойному щелчку мыши на строке URL, после чего откроется окно браузера

и отобразится стартовая страница GWT-приложения.

При выборе читателя Петров П.П. в окне браузера при нажатии кнопки «Получить список книг» отобразится следующая HTML-страница:

Сервис для отображения списка книг читателя

Выберите читателя:

Петров П.П. ▼

Получить список книг.

Список книг читателя Петров П.П.

Автор книги	Название книги	Прочитал
Толстой	Анна Каренина	Нет
Достоевский	Игрок	Да

Закреть

После закрытия диалогового окна можно задать имя другого читателя и получить список его книг.

Порядок выполнения лабораторной работы

1. Создайте и запустите шаблон GWT-проекта.
2. На основе файлов шаблона разработайте GWT-приложение, функциональность которого соответствует приложению, созданному в лабораторной работе № 3.
3. Выполните сборку и запустите GWT-приложение.
4. Сгенерируйте документацию с помощью Javadoc для клиентской и серверной частей приложения и просмотрите ее в браузере.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Скриншоты, иллюстрирующие работу приложения.
2. XML-файл приложения.
3. Текст документации, сгенерированный Javadoc.

Лабораторная работа № 9. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ WEB - ПРИЛОЖЕНИЯ

Цель работы: знакомство и использование фреймворка Mockito для модульного тестирования web-приложений

Установка и настройка фреймворка Mockito

Основная задача при тестировании сервлета – это проверка правильности работы методов `doGet` и `doPost`, которые ссылаются на контекст сервлета и параметры сеанса. Для того, чтобы исключить Tomcat из процесса тестирования и смоделировать среду передачи параметров сервлету, необходимо предоставить экземпляры классов `HttpServletRequest`, `HttpServletResponse`, `HttpSession` и другие. Фреймворк Mockito предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit тестов. Чтобы добавить Mockito в проект Eclipse, необходимо скачать последнюю или другую версию jar-архива, со страницы загрузки Mockito (<https://search.maven.org/artifact/org.mockito/mockito-core>) и загрузить его на свой жесткий диск. Затем надо щелкнуть правой кнопкой мыши на проекте и выбрать пункт "Properties". В появившемся окне (рис. 9.1) выбрать пункт "Java Build Path" и вкладку "Libraries". На вкладке "Libraries" нажать кнопку "Add External JARs" и перейти к jar файлу, который ранее был загружен. Нажать кнопку "Apply and Close", после чего библиотека Mockito будет добавлена в проект и доступна для использования.

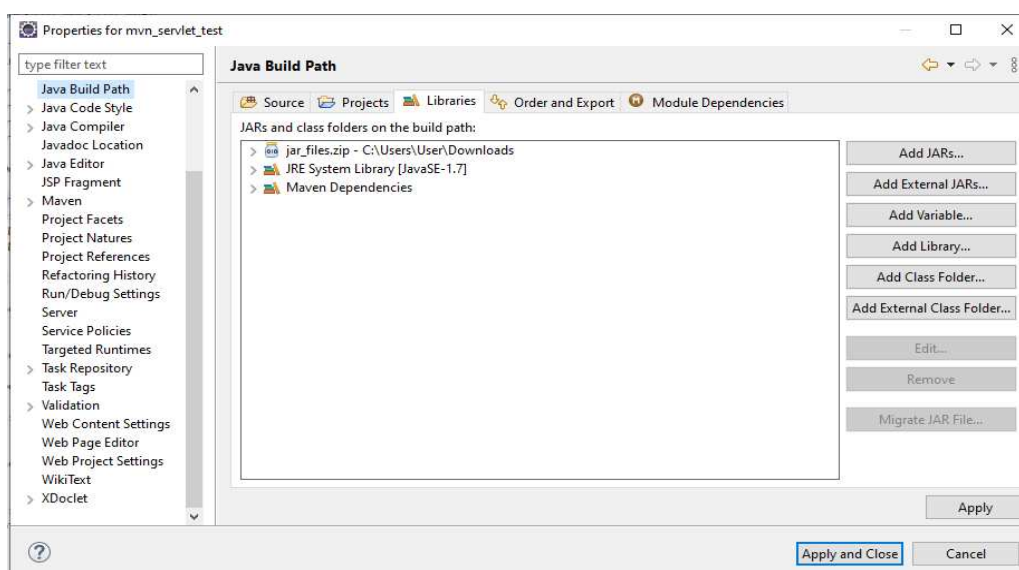


Рис. 9.1 Окно добавления библиотеки Mockito в проект

Для подключения компонентов Mockito в pom.xml файл проекта необходимо добавить зависимость. Например, для Jar-файла mockito-core-3.11.2 зависимости будут выглядеть так:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.11.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Mockito позволяет создать так называемый mock (объекты, которые заменяют реальный объект в условиях теста) для любого класса. Для этого объекта можно переопределить поведение желаемым образом, проконтролировать с нужной степенью детальности обращения к нему. Чтобы создать Mockito объект можно использовать либо аннотацию @Mock, либо методом mock. Например, объявить mock объекта HttpServletRequest можно так:

```
@Mock
```

```
HttpServletRequest request;
```

или

```
HttpServletRequest request = mock(HttpServletRequest.class);
```

Метод mock объекта возвращает значения по умолчанию: false для boolean, 0 для int, пустые коллекции, null для остальных объектов. Mock-объект будет идентифицирован как объект объявленного класса. Он будет принят любым методом или конструктором, которому требуется параметр такого типа. В методе setUp() перед началом теста (@Before) надо вызвать функцию MockitoAnnotations.initMocks(this) или MockitoAnnotations.openMocks(this), которая позволит инициализировать все объекты, помеченные аннотациями @Mock.

В тестах Mockito можно выделить две части: определение поведения методов и верификация. Задать поведение метода можно с помощью следующих функций Mockito:

Название функции	Описание функции
<code>thenReturn(T value)</code>	Задать возвращаемое значение
<code>thenThrow(Throwable... throwables)</code>	Задать исключение, которое будет брошено
<code>thenAnswer(Answer answer)</code>	Перехватить результаты
<code>doReturn(Object toBeReturned)</code>	Задать значение, которое будет возвращено заранее
<code>doThrow(Throwable... toBeThrown)</code>	Задать исключение для метода <code>mocked void</code>
<code>doAnswer(Answer answer)</code>	Перехват результатов заранее
<code>doCallRealMethod()</code>	Вызов определенного метода
<code>doNothing()</code>	Задать для метода <code>void</code> ничего не делать

Функции `when-thenReturn(value)` и `doReturn(value)-when` позволяют определить возвращаемое значение при вызове метода `mock` с заданными параметрами. Их различие заключается в том, что функция `doReturn(value)-when` не создает побочного эффекта и для нее не выполняется проверка на этапе компиляции типа возвращаемого значения. Если в описании функций будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернется последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Например,

```
when(request.getParameter("password")).thenReturn("password1", "password2");
```

Если нужно задать реакцию на любой вызов этого метода независимо от аргументов, то можно воспользоваться методом `Mockito.any()`:

```
when(request.getParameter(any())).thenReturn("");
```

Если метод имеет исключение, то реакция на его вызов может быть генерация исключительной ситуации:

```
when(request.getParameter("password")).thenReturn("password1").thenThrow  
(new IllegalArgumentException());
```

Здесь первый вызов метода вернёт значение "password1", а второй

выбросит исключение. Если требуется сгенерировать исключение для метода, который возвращает void, то надо воспользоваться функцией doThrow. Например, для метода void setText(String)

```
Mockito.doThrow(new RuntimeException()).when(mock).setText("abc");
```

Для блокировки исключения надо применить функцию doNothing:

```
Mockito.doNothing().doThrow(new NullPointerException()).when(mock).setText("abc");
```

Когда необходимо вызвать реальный метод из mock-объекта, то в этом случае нужно воспользоваться функцией doCallRealMethod(). Например, doCallRealMethod()when(mock).setText("abc");

Если нужно выполнить какую-то операцию или значение должно быть вычислено во время выполнения, то надо использовать функцию thenAnswer(Answer answer). Ответ(Answer) — это функциональный интерфейс, имеющий метод answer(..). Этот метод будет вызван при вызове метода mock-объекта. Например, нужно получить системное время при вызове метода getCurrentTime:

```
Mockito.when (mock.getCurrentTime() ).thenAnswer(() -> new Date() );
```

Здесь реализация метода answer описана в виде лямбда-выражения.

Верификация позволяет определить была ли выполнена проверка mock объекта и сколько раз. Она выполняется с помощью метода **verify(mock_object).method_call**, в котором указывается какая операция контролируется. Если проверка не выполнялась или выполнялась с другими параметрами, то это приведет к падению теста. Например, проверка факта однократного вызова метода на протяжении выполнения теста, выглядит так:

```
verify(request).getParameter("password");
```

Для проверки количества вызовов методов Mockito предоставляет следующие функции:

atLeast (int min) - не меньше min вызовов;

atLeastOnce () - хотя бы один вызов;

atMost (int max) - не более max вызовов;

times (int cnt) - cnt вызовов;

never () - вызовов не было.

Проверка, был ли вызван метод 2 раза выглядит так:

```
verify(request, times(2)).getParameter("password");
```

Разработка теста для сервлета с помощью фреймворка Mockito

В качестве примера рассмотрим описание теста для метода doPost сервера, реализующего ввод, контроль имени пользователя и пароля. В файле index.jsp определены следующие элементы интерфейса: поля ввода имени(*user*) и пароля(*password*), кнопка(*login*) и поле сообщения об ошибке(*error*).

```
<html>
  <head>
    <title>login form</title>
  </head>
  <body>
    <form method="post" action="Login">
      User:<input type="text" name="user" /><br/>
      Password:<input type="text" name="password" /><br/>
      <input type="submit" value="login" />
    </form>
    <p>Error: <output name="error"><% if (request.getAttribute("error") !=null)
      out.println(request.getAttribute("error")); %></output></p>
  </body>
</html>
```

Ниже приведен код метода doPost, который считывает с экранной формы имя пользователя и пароль, сравнивает их соответственно со значениями "12345" и "passw0rd". Если данные совпадают, то создаются два фрагмента данных (Cookie) для хранения имени пользователя и пароля, а затем генерируется html-страница с сообщением «Login successfull...». В противном случае в поле *error* записывается строка «ERROR» и выводится начальная страница index.jsp с сообщением об ошибке.

```
public void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException,
IOException {
    String name = req.getParameter("user");
    String pwd = req.getParameter("password");
    if (name.equals("12345") && pwd.equals("passw0rd")) {
        HttpSession session = req.getSession();
        session.setAttribute("user", name);
        Cookie ck1 = new Cookie("user", name);
        Cookie ck2 = new Cookie("pwd", pwd);
        res.addCookie(ck1);
        res.addCookie(ck2);
        PrintWriter out = res.getWriter();
        out.write("Login successfull...");
    }
    else {
        req.setAttribute("error", "ERROR");
    }
}
```

```

        RequestDispatcher rd = req.getRequestDispatcher("/index.jsp");
        rd.forward(req, res);
    }}

```

Первым этапом при разработке теста является выделение mock-объектов (заглушек), которые будут имитировать работу внешних экземпляров классов. Такими классами для приведенного выше метода `doPost` являются: `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `RequestDispatcher`. Mock-объекты этих классов должны быть объявлены в тесте и проинициализированы в методе `setUp()` до начала работы теста.

На втором этапе определяется состав контролируемых методов заглушек, которые будут использоваться при тестировании. В данном примере это `getParameter`, `getAttribute`, `setAttribute`, `getRequestDispatcher`, `getWriter`, `forward`. Для этих функций надо определить поведение, т.е. требуется указать как должен отреагировать метод при его вызове с заданными параметрами.

Третий этап решает задачу слежения за вызовами методов. Нужно убедиться в том, что тестируемый класс вызывает методы mock-объектов нужное число раз, в нужном порядке и с нужными параметрами.

С учетом указанной последовательности был разработан тестовый класс `LoginServletTest`, который приведен ниже:

```

package mvn_servlet_test;

import static org.mockito.Mockito.*;
import java.io.PrintWriter;
import java.io.StringWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletContext;
import javax.servlet.ServletConfig;
import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import demo.servlet.Login;

public class LoginServletTest extends TestCase {
    @Mock
    HttpServletRequest request,request1;
    @Mock
    HttpServletResponse response,response1;
    @Mock

```

```

HttpSession session;
@Mock
RequestDispatcher rd,rd1;

@Before
protected void setUp() throws Exception {
MockitoAnnotations.openMocks(this);

}

@Test
public void test() throws Exception {
    when(request.getParameter("user")).thenReturn("12345");
    when(request.getParameter("password")).thenReturn("password");
    when(request.getSession()).thenReturn(session);
    when(request.getRequestDispatcher("/index.jsp")).thenReturn(rd);

    when(request1.getParameter("user")).thenReturn("12345");
    when(request1.getParameter("password")).thenReturn("password");
    when(request1.getRequestDispatcher("/index.jsp")).thenReturn(rd1);
    when(request1.getAttribute("error")).thenReturn("ERROR");

    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);

    when(response.getWriter()).thenReturn(pw);
    Login s=new Login();
    s.doPost(request, response);

    verify(session).setAttribute("user", "12345");
    verify(rd).forward(request, response);
    verify(request).getRequestDispatcher("/index.jsp");
    String result = sw.getBuffer().toString().trim();
    assertEquals("Login successfull...", result);

    s.doPost(request1, response1);
    verify(request1).setAttribute("error", "ERROR");
    assertEquals("ERROR", request1.getAttribute("error"));
    verify(rd1).forward(request1, response1);
    verify(request1).getRequestDispatcher("/index.jsp");

}
}

```

Поскольку необходимо протестировать работу сервлета для правильного и неправильного задания пароля, то в тесте дважды вызывается метод doPost с разными запросами request и request1.

Файл с тестовым классом должен располагаться в каталоге /src/test/java/<имя проекта> и иметь имя, начинающееся с «Test» и заканчи-

вающееся «Test» или «TestCase», например LoginServletTest. Для запуска теста надо нажать правую кнопку мыши на имени проекта и в появившемся окне выбрать пункт Run As и вариант работы с проектом Maven install. По умолчанию будут запущены maven-compiler-plugin, компирующий исходные коды Web-приложения и теста, и maven-surefire-plugin, который автоматически выполнит тестовые классы. Результаты работы команды Maven install можно наблюдать в окне Console и в виде отчетов в форматах .txt и .xml, сохраняющиеся в директории /target/surefire-reports.

Порядок выполнения лабораторной работы

1. Установите и настройте в среде Eclipse фреймворк Mockito.
2. Разработайте и выполните тест с использованием mock-объектов для сервлета из лабораторной работы №3 или №5.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Исходные тексты тестов для тестирования сервлета
2. Отчеты по результатам тестирования.

Содержание

Лабораторная работа № 1. УСТАНОВКА И НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ И ИСПОЛНЕНИЯ Web-ПРИЛОЖЕНИЯ	2
Лабораторная работа № 2. СОЗДАНИЕ Web-ПРОЕКТА С ИСПОЛЬЗОВАНИЕМ СИСТЕМЫ СБОРКИ MAVEN	13
Лабораторная работа № 3. ПОСТРОЕНИЕ WEB-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ СЕРВЛЕТОВ	23
Лабораторная работа № 4. ИНТЕРНАЦИОНАЛИЗАЦИЯ WEB-ПРИЛОЖЕНИЙ	33
Лабораторная работа № 5. ПОСТРОЕНИЕ Web-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ JSP	39
Лабораторная работа № 6. АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ ПОЛЬЗОВАТЕЛЕЙ WEB-ПРИЛОЖЕНИЯ.....	50
Лабораторная работа № 7. ОРГАНИЗАЦИЯ ПЕРЕДАЧИ ДАННЫХ МЕЖДУ ЗАПРОСАМИ ПОЛЬЗОВАТЕЛЕЙ	56
Лабораторная работа № 8. РАЗРАБОТКА Web-ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ GWT.....	Ошибка! Закладка не определена.
Лабораторная работа № 9. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ WEB - ПРИЛОЖЕНИЯ	79

Редактор Э. К. Долгатов

Подписано в печать 00.00.22. Формат 60×84 1/16. Бумага офсетная.

Печать офсетная. Гарнитура «Times New Roman». Печ. л. 6,0.

Тираж 100 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5