Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

СОГЛАСОВАНО
Заведующий кафедрой
_____ М.В. Игнатенко
«28» декабря 2023 г.

СОГЛАСОВАНО
Декан факультета
_____ С.М. Босяков
«30» января 2024 г.
МП

Разработка Веб-приложений

Электронный учебно-методический комплекс для специальности:
1-31 03 08 «Математика и информационные технологии (по направлениям)».
Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии)

Регистрационный № 2.4.2-24 / 551

Автор:
Блинов И.Н., кандидат физико-математических наук, доцент.

Рассмотрено и утверждено на заседании Научно-методического совета БГУ

28.11.2024 г., протокол № 4.

Минск 2024

УДК 004.422:004.738.5(075.8)
Б 695

Утверждено на заседании Научно-методического совета БГУ
Протокол № 4 от 28.11.2024 г.

Решение о депонировании вынес:
Совет Механико-математического факультета.
Протокол № 6 от 30.01.2024 г.

Автор: Блинов Игорь Николаевич, кандидат физико-математических наук наук, доцент, доцент кафедры ВТиКМ ММФ БГУ

Рецензенты:

Лапицкая Н.В., доцент, зав. кафедрой Программного обеспечения информационных технологий факультета компьютерных систем и сетей УО «Белорусский государственный университет информатики и радиоэлектроники», кандидат технических наук,

Левчук В. Д., доцент кафедры Программного обеспечения информационных технологий факультета компьютерных систем и сетей УО «Белорусский государственный университет информатики и радиоэлектроники», кандидат технических наук.

Блинов, И.Н. Разработка Веб-приложений : электронный учебно-методический комплекс для специальности 1-31 03 08 «Математика и информационные технологии (по направлениям)» направления специальности 1-31 03 08-01 «Математика и информационные технологии (веб-программирование и интернет-технологии)» / И.Н. Блинов ; БГУ, Фак. механико-математический, Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2024. – 139 с. : ил. – Библиогр.: с. 135.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Разработка Веб-приложений» предназначен для студентов специальности 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 «Математика и информационные технологии (веб-программирование и интернет-технологии)». ЭУМК содержит тексты лекций, планы лабораторных занятий, перечень контрольных вопросов, списки рекомендованной литературы.

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Разработка Веб-приложений» предназначен для специальности 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии) .

Комплекс подготовлен в соответствии с требованиями Положения об учебно-методическом комплексе на уровне высшего образования, утвержденного Постановлением министерства образования Республики Беларусь от 26.07.2011 № 167.

Содержание разделов ЭУМК соответствует образовательным стандартам, структуре и тематике типового учебного плана.

Главные цели ЭУМК: помощь студентам в организации самостоятельной работы, повышение качества подготовки и усиление практико-ориентированности учебного процесса по дисциплине « Разработка Веб-приложений».

ЭУМК состоит из следующих разделов.

Теоретический. Включает аннотацию учебного пособия, написанного в соответствии с программой дисциплины. Материал данного пособия может быть использован для самостоятельной подготовки студентов к лабораторным занятиям, контрольным работам и зачёту в соответствиии с учебной программой Разработка Веб-приложений учреждения высшего образования по учебной дисциплине для специальности 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии) № УД – 11151 уч. [Электронный ресурс]. – Режим доступа: – Дата доступа: 10.06.2023. И в соответствиии с учебной программой учреждения высшего образования по учебной дисциплине Разработка Веб-приложений для специальности первой ступени высшего образования 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии) . № УД-11151/уч. [Электронный ресурс]. – Режим доступа:– https://elib.bsu.by/handle/123456789/294913 Дата доступа: 10.06.2023.

Практический. Содержит планы лабораторных работ. Данные материалы используются для подготовки к лабораторным занятиям и их организации, для самостоятельной работы над курсом.

Раздел контроля знаний представлен вопросами к зачёту.

Описаны формы диагностики и технология определения оценки по дисциплине с учетом текущей успеваемости.

Вспомогательный раздел включает рекомендуемую литературу и содержание учебной программы курса по отдельным темам, на основе которой построено изучение дисциплины и контроль знаний, вопросы к зачёту.

Учебная дисциплина «Разработка Веб-приложений» предназначен для студентов специальности 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии). В учебной дисциплине изучаются теоретические основы баз данных, рассматриваются вопросы анализа и проектирования информационных систем. Изучается работа с программными средствами автоматизации проектирования информационных систем.

Изучение дисциплины должно способствовать формированию у студентов основ алгоритмического мышления и представления о современных подходах к программному решению научных и прикладных задач, позволяет применить знания, полученные при изучении проектирования информационных систем, для практического решения задач проектирования, программирования и использования информационных систем, возникающих в различных областях промышленности и науки.

Цели и задачи учебной дисциплины

Дисциплина «Разработка Веб-приложений» имеет прикладную направленность.

Цель учебной дисциплины — формирование у студентов устойчивых теоретических знаний и практических навыков в области разработки и эксплуатации информационных систем, использования средств

автоматизированного проектирования и программных продуктов, реализующих функционирование и управление информационных систем.

Задачи учебной дисциплины

– получение знаний о языке Java;

– углубление знаний о концепциях, положенных в основу шаблонов проектирования программных интернет- систем;

– углубление знаний написания программного кода;

– формирование знаний, необходимых для более углубленного изучения технологий Jakarta EE.

Место учебной дисциплины в системе подготовки специалиста с высшим образованием.

Учебная дисциплина «Разработка Веб-приложений» относится к государственному компоненту.

Связи с другими учебными дисциплинами, включая учебные дисциплины компонента учреждения высшего образования, дисциплины специализации и др.

Учебная дисциплина «Разработка Веб-приложений» взаимосвязана с учебной дисциплиной «Технологии программирования».

Требования к компетенциям

Освоение учебной дисциплины «Разработка Веб-приложений» должно обеспечить формирование следующих компетенций:

базовые профессиональные компетенции:

БПК-3. Применять современные компьютерные математические системы для проведения вычислительного (компьютерного) эксперимента.

БПК-6. Применять современные технологии и базовые конструкции языков программирования для реализации алгоритмических прикладных задач и разработки веб-проектов.

БПК-9. Применять инновационные информационные технологии и современные языки программирования.

специализированные компетенции*:*

– СК-1. Осуществлять анализ контекста и поставленной проблемы, аргументированно выбирать оптимальный способ ее решения, согласовывать частичные проекты решения в общую согласованную архитектуру, выполнять реализацию проекта с учетом накопленных и поступающих данных.

В результате изучения дисциплины студент должен:

знать:

- основные этапы построения информационных систем.
- принципы функционирования и взаимодействия распределенных систем на платформе JEE.

- передовые технологии разработки в области веб-программирования
- правила использования документации и учебных материалов от авторов языка Java.

Структура учебной дисциплины

Учебная дисциплина «Разработка Веб-приложений» изучается в 4 семестре дневной формы получения высшего образования, в семестре заочной формы получения высшего образования для направления специальности 1-31 03 08 Математика и информационные технологии (по направлениям). Направления специальности: 1-31 03 08-01 Математика и информационные технологии (веб-программирование и интернет-технологии) :

Дисциплина изучается в 4-м семестре дневной и заочной форм обучения. На изучение учебной дисциплины «Разработка веб-приложений» отводится всего 102 часа, в том числе:

– 68 аудиторных часов, из них лекции – 34 часа, лабораторные занятия – 30 часов, управляемая самостоятельная работа – 4 часа.

Трудоемкость учебной дисциплины составляет 3 зачетных единицы.

Форма текущей аттестации по данной дисциплине:

– зачет в 4-м семестре.

Дисциплина изучается в 6-м семестре заочной форм обучения. На изучение учебной дисциплины «Разработка веб-приложений» отводится всего 54 часа, в том числе:

– 12 аудиторных часа, из них лекции – 6 часов, лабораторные занятия – 6 часов.

Трудоемкость учебной дисциплины составляет 2 зачетных единицы.

Форма текущей аттестации по данной дисциплине:

– зачет в 6-м семестре.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

В теоретическом разделе изложен краткий конспект лекций в соответствии с программой дисциплины.

Данное пособие ориентировано на студентов механико-математического факультета БГУ, изучающих современное программирование.

В первой части пособия рассмотрены основы серверных технологий на основе сервлетов.

Вторая часть пособия посвящена изучению JSP (*Java Server Pages*), технолонии приема и передачи данных от клиента к серверу.

Третья часть пособия освящает события, сессии, фильтры.

Четвертая часть пособия посвящена применению библиотек тегов.

1.1. Сервлеты

Приложение Java, запускаемое и выполняемое в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Сервлеты поддерживаются виртуальной машиной JVM, что предотвращает утечки памяти и обеспечивает функционирование garbage collection. Каждому клиенту сервлет выделяет независимый поток выполнения. Клиент посылает приложению HTTP-запрос, сервлет генерирует ответ и возвращает его клиенту в виде html-документа.

Сервлет:

— компонент приложений Jakarta Enterprise Edition;

— загружается application-сервером в контейнер;

— выполняется на стороне сервера;

— обрабатывает клиентские запросы;

— динамически генерирует ответы;

— находится в состоянии ожидания, если запросы отсутствуют;

— принимает\передает запросы от других сервлетов (Servlet chaining);

— поддерживает соединения с ресурсами.

Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Технология сервлетов является оболочкой протокола HTTP и поддерживает его как транспорт передачи данных от клиента серверу и обратно. Контейнер сервлетов поддерживает также протокол HTTPS (HTTP и SSL) для защищаемых запросов.

Сервлеты в промышленном программировании используются для:

— приема входящих данных от клиента;

— взаимодействия с бизнес-логикой системы;

— динамической генерации ответа клиенту.

Все сервлеты реализуют общий интерфейс **Servlet** из пакета **jakarta.servlet**. Для обработки HTTP-запросов в web можно воспользоваться в качестве базового класса абстрактным классом **HttpServlet**

Жизненный цикл сервлета начинается с его создания, инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов.

Завершение существования происходит при выгрузке его из контейнера.

Первым вызывается метод **init()**. Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе размещается код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **void service(HttpServletRequest request, HttpServletResponse response)** сервлета, вызываемый контейнером, а все параметры запроса упаковываются в экземпляр **request** интерфейса **HttpServletRequest**, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр **response** интерфейса **HttpServletResponse**, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода **service()**. Метод **service()** предназначен для одновременной обработки множества запросов.

При выгрузке приложения из контейнера, то есть по окончании жизненного цикла сервлета, вызывается метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Преимуществом сервлетов перед CGI или ASP является быстродействие, переносимость на различные платформы, использование объектно-ориентированного языка высокого уровня Java, который расширяется большим числом классов и программных интерфейсов.

Сервлеты поддерживаются серверами приложений WebSphere, Weblogic, WildFly, Oracle OC4J Server, Glassfish, Geronimo, Apache Tomcat, на основе контейнеров сервлетов tomcat, jetty, grizzly, glassfish и являются частью платформы JEE.

Сервлеты реализуют интерфейс **Servlet**, в котором кроме рассмотренных выше методов **service()**, **init()**, **destroy()** предусмотрена реализация метода **ServletConfig getServletConfig()** — он возвращает объект, содержащий параметры конфигурации сервлета и дает доступ к среде выполнения.

При разработке сервлетов в качестве суперкласса в большинстве случаев используется не интерфейс **Servlet**, а абстрактный класс **HttpServlet**, отвечающий за обработку запросов HTTP.

Метод **service()** класса **HttpServlet** служит диспетчером для других методов, каждый из которых обрабатывает методы доступа к соответствующим ресурсам. В спецификации HTTP определены методы: **GET**, **HEAD**, **POST**, **PUT**, **DELETE**, **OPTIONS** и **TRACE**. Наиболее часто употребляются методы **GET** и **POST**, передающие на сервер запросы, а также параметры для их выполнения.

В задачу метода **service()** класса **HttpServlet** входит анализ полученного через запрос метода доступа к ресурсам и вызов метода со сходным именем, где перед именем добавляется префикс **do**: **doGet()**, **doPost()**, **doHead()**, **doPut()**, **doDelete()**, **doOptions()** и **doTrace()**. Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

Метод **GET** (method="GET") используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер (2048 символов). Этот метод является основным методом взаимодействия браузера клиента и веб-сервера.

Метод **POST** используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка **Content-Type** и/или включены в URI запроса. При использовании метода **POST** под URI подразумевается ресурс, который будет обрабатывать запрос.

Метод **PUT** схож с методом **POST** за тем исключением, что здесь URI подразумевает ресурс, который будет создан или сохранен на сервере в результате выполнения **PUT**-запроса.

Метод **DELETE** предназначен для удаления целевого ресурса.

Оба эти действия на некоторых серверах могут запрещаться из-за угрозы внутренней безопасности.

Метод **HEAD** предполагает возврат сервером такого же ответа, как и при использовании **GET**, но без тела ответа. Метод обычно используется для того, чтобы проверить существование ресурса либо узнать, изменился ли запрашиваемый ресурс с момента последнего обращения.

Метод **OPTIONS** должен возвращать информацию о возможностях веб-сервера или параметрах соединения для конкретного ресурса.

Метод **TRACE** возвращает клиенту запрос в том виде, в каком он пришел на сервер — используется для отладки, определяя заголовки, добавляемые промежуточными серверами, а также для тестирования настроек соединения.

Методы **HEAD**, **OPTIONS** и **TRACE**, как правило, реализуются сервером веб-приложения прозрачно для программиста и не требуют какой-либо специальной обработки. Поэтому обычно при разработке веб-приложений они не рассматриваются.

Методы **GET** и **HEAD**, по принятым соглашениям, не должны иметь другого назначения, кроме как передача информации клиенту — они должны быть «безопасными» («safe»). То есть в результате их выполнения на сервере не должно происходить никаких «побочных эффектов», которые повлияют на состояние ресурсов сервера — таких, как создание, изменение, удаление данных. Также подразумевается, что запросы с «безопасными» методами могут быть выполнены браузером клиента в автоматическом режиме, без специального разрешения пользователя. Так как ссылки через элемент <a> в коде HTML по умолчанию подразумевают использование браузером **GET**-запросов, то довольно часто можно встретить нарушение «безопасности» **GET**-запросов из-за использования элемента <a> для выполнения каких-либо действий, изменяющих состояние сервера, например, ссылки вида:

```
<a href="http://example.com/do?action=delete">Delete database</a>
```

Далеко не всегда такое нарушение влечет негативные последствия, однако так как инфраструктура глобальных сетей предполагает соблюдение соглашения о «безопасности» **GET**-запросов, то при взаимодействии со следующими факторами могут возникать негативные последствия.

1. **Кэширующие прокси.** Так как **GET**-запрос (в отличие от **POST**) может быть кэширован промежуточным кэширующим прокси-сервером, то при обработке запроса прокси может не выполнить запрос к целевому серверу, а сразу вернуть результат. Таким образом, если запрос подразумевал какое-то действие над данными на сервере, то это действие не будет выполнено.
2. **Предзагрузка.** Некоторые браузеры для ускорения навигации выполняют загрузку страниц и других ресурсов по ссылкам еще до того, как пользователь выполнил переход по этим ссылкам. В этом случае при нарушении «безопасности» браузер выполнит запрос, который не был инициирован пользователем. Например, если на странице расположены ссылки, удаляющие какие-то объекты через **GET**-запросы, то при открытии этой страницы браузер может вызвать удаление всех этих объектов.

3. **Поисковые системы и другие клиенты, действующие в автоматическом режиме.** Разнообразные автоматические клиенты не могут отличить запросы, которые выполняют какие-либо действия от запросов, которые просто возвращают информацию. Поэтому они обычно ориентируются по методу запроса: **GET**-запросы выполняются и используются для дальнейшего анализа, а **POST**-запросы игнорируются. При нарушении «безопасности» **GET**-запросов возникает ситуация, аналогичная предзагрузке: выполняются те действия, которые при «нормальном» взаимодействии с приложением не должны были бы выполняться.
4. **Нарушение авторизации**. Если выполнение действия предполагает наличие определенных полномочий, то использование **GET**-запроса для этого предоставляет злоумышленнику возможность воспользоваться привилегиями другого пользователя (например, путем размещения ссылки с нужным **GET**-запросом в <img> элементе на произвольном сайте, который может посетить пользователь, обладающий нужными полномочиями с идентификацией через cookies) и тем самым выполнить неавторизованное действие.

Поэтому важно понимать различия между методами **GET** и **POST** и использовать **GET** только там, где выполняется только непосредственная передача данных без выполнения активных действий.

Методы **GET**, **HEAD**, **PUT** и **DELETE** должны быть идемпотентными («idempotence») в том смысле, что повторное (более одного раза) выполнение запросов с помощью этих методов будет иметь тот же эффект, что и однократное выполнение.

По умолчанию параметры передаются в запрос в формате:

?name1=value1&name2=value2

Однако форматы упаковки параметров могут быть самые разные, например, в случае передачи файлов с использованием формы

enctype="multipart/form-data"

Также часто вместо формата представления параметров вида

/article.jsp?id=1234&page=42

может быть использована так называемая Friendly URL форма, более понятная для человека, не являющегося программистом, и может быть представлена подобным образом:

/article/1234/page/4

либо

/article_1234/page_4

Однако такая форма представления параметров обычно требует дополнительной работы со стороны программиста.

Механизмами протокола HTTP не предусмотрено сохранение информации о своем клиенте. Однако веб-приложение может требовать информации о клиенте, особенно такое, реализация которого предусматривает аутентификацию клиента без необходимости ее подтверждения при смене страниц. Кроме поддержки распределенной сессии, о которой разговор пойдет в главе 17, существуют и примитивные механизмы получения сведений о клиенте:

— файлы cookie — текстовые файлы, ассоциированные с приложением и сохраняемые на компьютере клиента.

— добавление в строку GET-запроса дополнительных параметров.

— скрытые HTML-теги вида

<input type="hidden" name ="command" value ="Login"/>

которые необходимо добавлять в каждую страницу приложения для сохранения целостности ее архитектуры.

В следующем примере приведен готовый к выполнению, но не к практическому применению, шаблон сервлета:

```
// # 1 # простейший сервлет # FirstServlet.java
package by.bsu.first.servlet;
import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/FirstServlettest")// test -> clazz
public class FirstServlet extends HttpServlet {
  public FirstServlet() {
  }
  public void init() throws ServletException {
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
```

```
        response.setContentType("text/html");
         response.getWriter().print("This is " + this.getClass().getName() +
                                                    ", using the GET method");
      }
     protected void doPost(HttpServletRequest request, HttpServletResponse
response)
                          throws ServletException, IOException {
        response.setContentType("text/html");
         response.getWriter().print("This is " + this.getClass().getName() + ",
                                                     using the POST method");
      }
     public void destroy() {
         super.destroy(); // Just puts "destroy" string in log
     }
   }
```

С версии Servlet API 3.0 регистрация сервлетов в файле-дескрипторе **web.xml** необязательна. Конфигурация определяется с помощью аннотации **@WebServlet** и в расширенном виде с указанием имени сервлета в контексте:

```
@WebServlet(name = "FirstServletname", urlPatterns = { "/FirstServlettest", "*.do"
})
public class FirstServlet extends HttpServlet {
}
```

Аннотация **@WebServlet** может содержать и другие методы-члены, о чем будет сказано ниже. Также аннотированию подлежат методы реализации функциональности сервлета.

Практика включения HTML-кода в код сервлета не считается хорошей, так как эти действия «уводят» сервлет от его основной роли — контроллера приложения. Это приводит к росту объема кода сервлета, который на определенном этапе становится неконтролируемым и реализует вследствие этого антишаблон «Волшебный сервлет». Приведенный выше сервлет имеет признаки антишаблона, так как содержит метод **print()** для формирования кода HTML. Сервлет должен использоваться только для контроля реализации бизнес-логики приложения и обязан быть отделен как от непосредственного формирования текста ответа на запрос, так и от данных, необходимых для этого. Признаки наличия антишаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Сервлет является компонентом веб-приложения, который будет называться **FirstProject** и размещаться в папке **/WEB-INF/classes** проекта.

1.1.1. Запуск контейнера сервлетов и размещение проекта

Здесь и далее применяется веб-сервер Apache Tomcat в качестве обработчика страниц JSP и сервлетов. Последняя версия может быть загружена с ресурса **https://tomcat.apache.org/**.

При установке Tomcat предложит значение порта по умолчанию **8080**, но во избежание конфликтов с иными Application Server рекомендуется присвоить другое значение, например **8089**.

Ниже приведены необходимые действия по запуску сервлета из предыдущего примера с помощью сервера Tomcat, который установлен в каталоге **/Apache Software Foundation/Tomcat[***версия***]**. В этом же каталоге размещаются следующие подкаталоги:

/bin — содержит файлы запуска монитора и контейнера сервлетов вида **tomcat[***номер***].exe**, **tomcat[***номер***]w.exe** и некоторые необходимые для этого библиотеки;

/lib — содержит библиотеки служебных классов, в частности Servlet API и JSP API; используемые внешние библиотеки (если они есть), упакованные в JAR-файлы;

/conf — содержит конфигурационные файлы, в частности, конфигурационный файл сервера **server.xml**;

/logs — в этот каталог записываются log-файлы, инициированные сервером;

/webapps — в этот каталог помещаются папки с веб-приложениями, содержащие в свою очередь сервлеты и другие компоненты конкретного приложения.

В каталог **/webapps** необходимо поместить папку **/FirstProject** с вложенным в нее сервлетом **FirstServlet**. Кроме того, папка **/FirstProject** должна содержать каталог **/WEB-INF**, в котором помещаются подкаталоги:

/classes — содержит class-файл сервлета **by.bsu.first.servlet.FirstServlet**;

/src — содержит исходный файл сервлета **FirstServlet.java** (опционально); а также **web.xml** — дескриптор доставки (развертывания) приложения располагается в каталоге **/WEB-INF**.

В файле **web.xml** для версий Servlet API, не превышающих 2.5, необходимо прописать имя и путь к сервлету. Кроме того в дескрипторном файле можно определять параметры инициализации, MIME-типы, mapping сервлетов и JSP, стартовые страницы и страницы с сообщениями об ошибках, а также параметры для безопасной авторизации и аутентификации. Этот файл можно сконфигурировать так, что путь к сервлету в браузере не будет совпадать с истинным именем класса сервлета. Например:

 # 2 # простейший дескрипторный файл # web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
  <display-name>FirstProject</display-name>
  <servlet>
                <display-name>FirstServletdisplay</display-name>
                <servlet-name>FirstServletname</servlet-name>
                <servlet-class>by.bsu.first.servlet.FirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
                <servlet-name>FirstServletname</servlet-name>
                <url-pattern>/FirstServlettest</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/resources/*</url-pattern>
  </servlet-mapping>
  <session-config>
                <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Здесь указано имя сервлета **FirstServletname**, путь к откомпилированному классу сервлета **FirstServlet.class**, а также URL-имя сервлета **FirstServlettest**, по которому происходит его вызов.

Для версии сервлетов 3.0 и выше теги мэппинга не нужны, так как эта информация определяется аннотацией.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
    version="5.0">
      <display-name>FirstProject</display-name>
      <session-config>
        <session-timeout>30</session-timeout>
      </session-config>
      <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
      </welcome-file-list>
    </web-app>
```

Таким образом, требуется выполнить действия:

1) компиляцию сервлета с указанием в **–cp** пути к архиву **servlet-api.jar**;
2) полученный файл класса **FirstServlet.class** поместить в каталог **/FirstProject/ WEB-INF/classes/by/bsu/first/servlet**;
3) в каталог **/FirstProject/WEB-INF** поместить файл конфигурации **web.xml**;
4) в каталог **/FirstProject** поместить файл стартовой JSP-страницы **index.jsp**;
5) разместить папку **/FirstProject** в каталог **/webapps** сервера Tomcat;
6) стартовать Apache Tomcat;
7) запустить браузер и ввести адрес

   http://localhost:8080/FirstProject/FirstServlettest

   При обращении к сервлету из другого компьютера вместо **localhost** следует указать IP-адрес или имя компьютера;
8) если вызывать сервлет из **index.jsp**, то тег **FORM** должен выглядеть следующим образом:

   <form action="FirstServlettest">
   <input type="submit" value="Execute"> </form>

   Для запуска проекта в браузере набирается строка

   http://localhost:8080/FirstProject/index.jsp

или

   http://localhost:8080/FirstProject/

Сервлет будет вызван из JSP-страницы по URL-имени **FirstServlettest**, и в результате в браузер будет выведено:

Рисунок 1 - *Вывод сервлета после вызова метода doGet()*

1.1.2. Простая JSP–страница

Технология Java Server Pages (JSP) обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, так как дизайнерам нет необходимости знать, как работать с динамическими данными.

JSP-код состоит из специальных тегов и выражений, которые указывают контейнеру соответствие между тегами и java-кодом для генерации сервлета или его части. Таким образом поддерживается документ, который одновременно содержит и статическую страницу, и теги Java, управляющие страницей. Статические части HTML-страниц посылаются в виде строк в метод **write()**. Динамические части включаются прямо в код сервлета. С момента формирования ответа сервера страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.

Чтобы создать простейшую JSP, достаточно взять HTML-страницу и заменить расширение **html** на **jsp**. Только в этом случае для запуска страницы необходим специальный application server с контейнером сервлетов и особое размещение самой страницы.

1.1.3. Взаимодействие сервлета и JSP

Страницы JSP и сервлеты никогда не следует использовать в информационных системах друг без друга. Причиной являются принципиально различные роли, которые играют данные компоненты в приложении. Страница JSP ответственна за формирование пользовательского интерфейса и отображение информации, переданной с сервера. Сервлет выполняет роль контроллера запросов и ответов, то есть принимает запросы от всех связанных с ним JSP-страниц, вызывает соответствующую бизнес-логику для их (запросов) обработки и в зависимости от результата выполнения решает, какую JSP поставить этому результату в соответствие.

При необходимости расширения функциональности системы *не следует* создавать дополнительные сервлеты. Сервлет в приложении должен быть *один*. При создании нового учебного приложения во избежание путаницы всегда следует создавать новый проект.

Для демонстрации взаимодействия JSP-страниц и сервлета будет решена задача определения времени между загрузкой страницы в браузер и нажатием кнопки на этой же странице.

Страница JSP с последующим вызовом другой JSP-страницы, отображающей результаты выполнения запроса.

 # 3 # страница JSP с вызовом сервлета # index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>JSP Timing</title>
  </head>
  <body>
    <h5>Счетчик времени от запуска приложения до нажатия кнопки</h5>
    <jsp:useBean id="calendar" class="java.util.GregorianCalendar" />
    <form name="Simple" action="timeaction" method="POST">
      <input type="hidden" name="time" value="${calendar.timeInMillis}" />
      <input type="submit" name="button" value="Посчитать время" />
    </form>
  </body>
</html>
```

В результате запуска стартовой страницы проекта в браузер будет выведено:

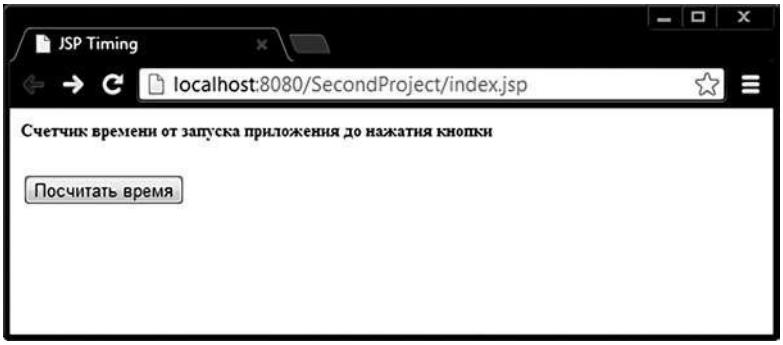


Рисунок 2 - *Стартовая страница приложения*

Кодировка для символов кириллицы задана с помощью директивы **page**. Action-тег **useBean** используется для создания объекта класса **GregorianCalendar** в области видимости JSP. Сервлет **TimeServlet** вызывается методом **POST**. Ему передается с переменной **calendar** информация о времени выполнения страницы.

```
// # 4 # простой контроллер # ServletTiming.java
package by.bsu.second.servlet;
import java.io.IOException;
import java.util.GregorianCalendar;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/timeaction")
public class TimeServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse
response)
                                          throws ServletException, IOException {
     processRequest(request, response);
   }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse
response)
                                              throws ServletException,
IOException {
     processRequest(request, response);
   }
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
                                          throws ServletException, IOException {
     GregorianCalendar gc = new GregorianCalendar();
     String timeJsp = request.getParameter("time");
     float delta = ((float) (gc.getTimeInMillis()-Long.parseLong(timeJsp)))/1_000;
     request.setAttribute("res", delta);
     request.getRequestDispatcher("/jsp/result.jsp").forward(request, response);
   }
}
```

Обмен информацией между JSP и сервлетом чаще всего осуществляется с помощью атрибутов объектов **HttpServletRequest**, **HttpSession**, **HttpServletContext**. Вызов **result.jsp** из сервлета в данном случае производится методом **forward()** интерфейса **RequestDispatcher**.

Для вызова JSP по относительному пути применяется метод **forward()**, для обращения к JSP по абсолютному пути используется метод **sendRedirect()**. Отличие этих методов состоит в том, что с методом **forward()** передается уже существующий объект запроса **request**, а при вызове метода **sendRedirect()** формируется новый запрос. Информацию в последнем случае следует передавать с другими объектами. К тому же метод **forward()** срабатывает быстрее.

 # 5 # страница, вызванная сервлетом # result.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Result Page</title>
  </head>
  <body>
    <p>Кнопка нажата через ${res} сек </p>
  </body>
</html>
```

После обращения к сервлету и выполнения им действий по созданию атрибута объекта **request** будет вызвана страница **result.jsp**, интерпретирующая результат в виде:

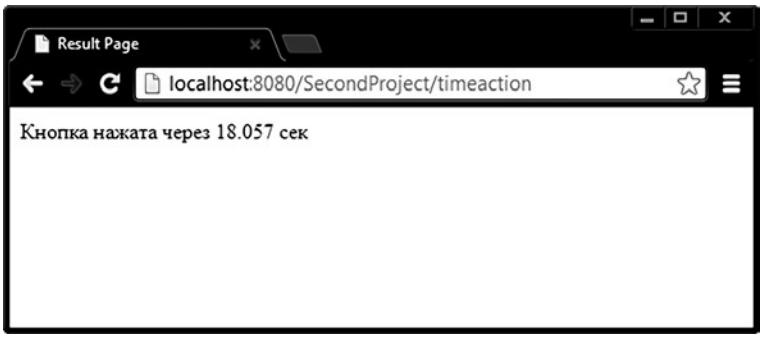


Рисунок 3 - *Вывод информации страницей result.jsp*

В данном коде используется Expression Language (EL) для доступа к атрибутам запроса в виде **${calendar.timeInMillis}** и **${res}**.

Кроме сокращенной нотации вызова методов с префиксом **get** по умолчанию можно использовать полную форму вызова метода с помощью EL, а именно: **${calendar.getTimeInMillis()}** или **${calendar.isLeapYear(2021)}**. Методы, которые не возвращают значение, вызвать нельзя.

1.1.4. Интерфейс ServletContext

Приложению ставится в соответствие единственный экземпляр, ассоциированный с контейнером сервлетов, реализующий **ServletContext**. Контекст выполнения сервлета предоставляет средства для общения с application-сервером и позволяет получать информацию о среде выполнения, а также использовать ресурсы совместно с другими объектами приложения. В частности, можно добавить/извлечь/удалить атрибуты, извлечь параметры контекста или записать информацию в log-файл. Получить ссылку на объект **ServletContext** можно вызовом метода **getServletContext()** на экземпляре сервлета.

Ряд методов предназначен для управления атрибутами, с помощью которых передается информация между различными компонентами приложения (JSP, сервлетами):

void setAttribute(String name, Object value) — добавляет атрибут;

Object getAttribute(String name) — получает значение атрибута по имени;

Enumeration getAttributeNames() — получает список имен атрибутов;

void removeAttribute(String name) — удаляет атрибут из контекста.

Другими словами, используя объект, реализующий **ServletContext**, можно связывать объекты с экземплярами сервлета, сессии и запроса.

Приложению могут быть заданы параметры инициализации (параметры контекста), получить доступ к которым можно с помощью методов:

String getInitParameter(String paramName) — извлечение параметра контекста по имени;

Enumeration getInitParameterNames() — извлечение набора имен параметров контекста, например, из кода метода сервлета

String uploadPath = **this**.getServletContext().getInitParameter("**upload_path**");

Для получения значения, необходимо, чтобы дескрипторный файл приложения **web.xml** содержал тег **<context-param>** со значением и именем параметра в виде:

```
<context-param>
  <param-name>upload_path</param-name>
  <param-value>C:\\upload</param-value>
</context-param>

<context-param>
  <param-name>admin_mail</param-name>
  <param-value>blinov@gmail.com</param-value>
</context-param>
```

Экземпляр **ServletContext** может применяться для обмена информацией между сервлетами приложения. Но в большей части промышленных приложений применяется односервлетная модель, поэтому использовать объект контекста следует только для хранения информации о конфигурации приложения или общей информации для всех клиентов.

Следующие методы позволяют получить из контекста сервлета базовую информацию:

String getContextPath() — возвращает путь к корню веб-приложения, находящегося в папке контейнера сервлетов и используется для доступа к jsp и другим компонентам веб-приложения;

String getRealPath(String path) — определение истинного маршрута файла относительно каталога, в котором сервер хранит текстовые и медиа файлы;

```
String path= this.getServletContext().getRealPath("/");
FileReader reader = new FileReader(path + "text/data.txt");
int code = reader.read();
char ch = (char)code;
request.setAttribute("code_char", code);
request.setAttribute("symbol", ch);
```

String getServerInfo() — возвращает имя и номер версии контейнера сервлетов;

String getServletContextName() — возвращает имя веб-приложения, заданное в дескрипторе приложения в теге **<display-name>**;

void log(String msg) — запись лога с сообщением **msg**;

void log(String msg, Throwable t) — запись лога с текстовым сообщением **msg** и сообщением, извлекаемым из исключения.

Интерфейс ServletConfig

Данный объект ассоциируется с самим сервлетом.

Параметры инициализации сервлета содержатся в экземпляре **ServletConfig**. Получить доступ к нему можно методом **getServletConfig()**, вызываемым на экземпляре сервлета.

Методы интерфейса:

Enumeration getInitParameterNames() — получение списка имен параметров инициализации сервлета;

String getInitParameter(String name) — определение значения конкретного параметра по его имени;

String getServletName() — определение имени сервлета, с которым связан текущий объект **ServletConfig**;

String getServletContext() — получение экземпляра контекста.

Чтобы задать параметры инициализации сервлета **EMailServlet**, который будет рассмотрен позже, необходимо в тег <**servlet**> дескрипторного файла **web.xml** вложить тег <**init-param**> с описанием имени и значения параметра в виде:

```
    <servlet>
  <servlet-name>EMailServlet</servlet-name>
  <servlet-class>by.bsu.mail.servlet.EMailServlet</servlet-class>
  <init-param>
    <param-name>mail.smtps.host</param-name>
    <param-value>smtp.gmail.com</param-value>
  </init-param>
  <init-param>
    <param-name>mail.smtp.port</param-name>
    <param-value>465</param-value>
  </init-param>
  <init-param>
    <param-name>smtps.auth.user</param-name>
    <param-value>blinov@gmail.com</param-value>
  </init-param>
  <init-param>
    <param-name>smtps.auth.pass</param-name>
    <param-value>real_password</param-value>
  </init-param>
  <init-param>
    <param-name>mail.transport.protocol</param-name>
    <param-value>smtps</param-value>
  </init-param>
</servlet>
```

или для версии 3.0 и выше:

```
@WebServlet(
   urlPatterns = {"/mailservlet"},
   initParams = {
      @WebInitParam(name = "mail.smtps.host", value = "smtp.gmail.com"),
      @WebInitParam(name = "mail.smtp.port", value = "465"),
      @WebInitParam(name = "smtps.auth.user", value = "blinov@gmail.com"),
        @WebInitParam(name = "smtps.auth.pass", value = "real_password"),
  @WebInitParam(name = "mail.transport.protocol", value = "smtp"),
  @WebInitParam(name = "mail.host", value = "smtp.gmail.com"),
  @WebInitParam(name = "mail.smtp.auth", value = "true"),
```

```
        @WebInitParam(name = "mail.smtp.socketFactory.port", value = "465"),
        @WebInitParam(name = " mail.smtp.socketFactory.class", value = "javax.net.ssl.SSLSocketFactory"),
        @WebInitParam(name = " mail.smtp.socketFactory.fallback", value = "false"),
        @WebInitParam(name = " mail.smtp.quitwait", value = "false")

          }
    )
    public class MailServlet extends HttpServlet {
    }
```

Тогда для доступа к параметрам инициализации сервлета и их дальнейшего использования в приложении, например, в коде метода **init()** сервлета следует применить:

```
    ServletConfig config = this.getServletConfig();
// определение набора имен параметров инициализации
Enumeration<String> e = config.getInitParameterNames();
while(e.hasMoreElements()){
   // определение имени параметра инициализации
   String name = e.nextElement();
   // определение значения параметра инициализации
   String value = sc.getInitParameter(name);
}
```

или

```
    this.getInitParameterNames()
      .asIterator()
      .forEachRemaining(o -> logger.log(Level.INFO, o + "=" + getInitParameter(o)));
```

1.1.5. Интерфейс HttpServletRequest

Информация от клиента веб-приложения отправляется серверу в виде объекта запроса типа **HttpServletRequest**. Данный интерфейс является производным от интерфейса **ServletRequest**. Используя методы интерфейса **ServletRequest**, можно получить: информацию о сервлете, доступ к сессии и параметрам запроса, управление атрибутами и кодировками, а также детали протокола HTTP.

При обращении к серверу с запросом от клиента передаются параметры и их значения. Для разбора параметров и извлечения их значений применяются методы:

String getParameter(String name) — определение значения параметра по его имени или **null**, если параметр с таким именем не задан;

String[] getParameterValues(String name) — определение всех значений параметра по его имени;

Enumeration getParameterNames() — определение ссылки на список имен всех параметров через объект типа **Enumeration**;

getParameterMap() —

String getCharacterEncoding() — определение символьной кодировки запроса;

void setCharacterEncoding(String code) — задание символьной кодировки запроса;

String getContentType() — определение MIME-типа (Multipurpose Internet Mail Extension) пришедшего запроса;

String getProtocol() — определение названия и версии протокола;

String getServerName(), **getServerPort()** — определение имени сервера, принявшего запрос, и порта, на котором запрос был принят сервером соответственно;

String getRemoteAddr(), **getRemoteHost()** — определение IP-адреса клиента, от имени которого пришел запрос, и его имени соответственно;

String getRemoteUser() — определение имени пользователя, выполнившего запрос;

String getQueryString() — извлечение строки HTTP-запроса;

String getMethod() — определение имени метода доступа к ресурсам, на основе которого построен запрос;

ServletInputStream getInputStream(), **BufferedReader getReader()** — получение ссылки на поток, ассоциированный с содержимым полученного запроса. Первый метод возвращает ссылку на байтовый поток **ServletInputStream**, а второй — на объект **BufferedReader**. В результате можно прочитать любой байт из полученного объекта-запроса. Если метод **getReader()** был вызван после вызова **getInputStream()** для этого запроса, то генерируется исключение **IllegalStateException** и наоборот.

Непосредственно в интерфейсе **HttpServletRequest** объявлен ряд методов, позволяющих связывать внешние ресурсы с запросом:

void setAttribute(String name, Object ob) — установка значения атрибута компонента, являющегося внутренним параметром для передачи информации между компонентами приложения, например, от сервлета к странице JSP или другому сервлету;

Enumeration getAttributeNames() — извлечение перечисления имен атрибутов;

Object getAttribute(String name) — извлечение значения переданного атрибута по имени;

void removeAttribute(String name) — удаление атрибута по имени;

Cookie[] getCookies() — извлечение массива cookie, полученного с запросом. Файл cookie — маленький символьный файл, сохраняемый приложением на стороне клиента;

Ниже рассматривается применение некоторых методов интерфейса **HttpServletRequest** для получения данных о запросе, посылаемом методом **GET** и генерации ответа клиенту. Вызов методов в сервлете **FirstServlet** проекта **FirstProject** на объекте **request.**

```
PrintWriter out = response.getWriter();
out.println(request.getMethod());
out.println(request.getRequestURL());
out.println(request.getProtocol());
out.println(request.getRemoteAddr());
out.println(request.getContextPath());
out.println(request.getScheme());
```

и передача результатов в браузер с помощью потока **PrintWriter** дадут следующие результаты:

Рисунок 4 - *Извлечение информации из запроса*

Из запроса также можно извлечь информацию о заголовке запроса, которая оказывается достаточно исчерпывающей:

```
Enumeration<String> names = request.getHeaderNames();
while (names.hasMoreElements()) {
   String headerName = names.nextElement();
   String value = request.getHeader(headerName);
   out.println(headerName + "=" + value);
}
```

Последний пример следует попробовать вызывать на разных компьютерах с различными конфигурациями, используя несколько браузеров, чтобы оценить выводимую информацию.
Результат выполнения фрагмента кода может выглядеть:

referer=http://localhost:8089/tom2_war_exploded/index.jsp
cookie=JSESSIONID=C5AFB1AF7C0147279C2AE63179C745A4; Idea-e935d2c9=5fe8e02c-7afc-4206-b75f-f8aff3c45f22
accept-language=ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3
upgrade-insecure-requests=1
host=localhost:8089
connection=keep-alive
accept-encoding=gzip, deflate
accept=text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0) Gecko/20100101 Firefox/65.0

Таким образом, при обращении клиента по адресу URL контейнер сервлета перехватывает запрос и вызывает метод **doGet()** или **doPost()**. Эти методы вызываются после конфигурации и создания объектов, реализующих интерфейсы **HttpServletRequest**, **HttpServletResponse**. Задача методов **doGet()** и **doPost()** — взаимодействие с HTTP-запросом клиента и создание HTTP-ответа, основанного на данных запроса и результатах взаимодействия с бизнес-логикой.

1.1.6. Интерфейс HttpServletResponse

Генерируемые сервлетами данные пересылаются серверу-контейнеру с помощью объектов, реализующих интерфейс **ServletResponse**, а сервер, в свою очередь, формирует и пересылает ответ клиенту, инициировавшему запрос.

В интерфейсе **HttpServletResponse**, наследующем интерфейс **ServletResponse**, есть несколько важных методов:

void setContentType(String type) — установка MIME-типа генерируемых документов;

void addCookie(Cookie cookies) — добавление файла cookie к объекту ответа для пересылки на клиентский компьютер;

void sendError(int sc, String msg) — переход на error-page с сообщением о возникших ошибках, где **sc** — код ошибки, **msg** — текстовое сообщение;

void sendRedirect(String location) — переход по указанному адресу, без передачи экземпляра **request**.

Можно получить ссылки на потоки вывода одним из двух методов:

ServletOutputStream getOutputStream() — извлечение ссылки на поток **ServletOutputStream** для передачи бинарной информации;

PrintWriter getWriter() — извлечение ссылки на поток типа **PrintWriter** для передачи символьной информации;

Если метод **getOutputStream()** уже был вызван для этого ответа, то генерируется **IllegalStateException**. Обратное также верно.

```
httpResponse.setHeader("Cache-Control","no-cache, no-store, must-revalidate");
response.addHeader("Cache-Control", "post-check=0, pre-check=0");
httpResponse.setHeader("Pragma","no-cache");
httpResponse.setDateHeader ("Expires", 0);
```

1.1.7. Атрибуты и параметры

Какие объекты и каким образом могут получать, передавать, предоставлять информацию? Ответ представлен в таблице:

	Атрибут	**Параметр**
Носитель	page (PageContext) request(HttpServletRequest) session (HttpSession) application (ServletContext)	context (инициализация) servlet (инициализация) request (запрос)
Метод установки	setAttribute(String name, Object value)	добавить параметры можно только к request
Метод извлечения	getAttribute(String name)	getInitParameter(String name) getParameter(String name)
Возвращаемый тип	Object	String и String[]

Рисунок 5 - *Таблица носителей информации*

Чаще всего для передачи информации между элементами системы используются объекты классов **HttpServletRequest** и **HttpSession**.

1.1.8. Многопоточность и электронная почта

Распределенное приложение может быть эффективным только в случае, если оно способно принимать информацию от физически удаленных клиентов и производить ее обработку максимально быстро. Увеличить скорость работы веб-приложения позволяет применение многопоточных режимов функционирования частей приложения.

Рассылка электронной почты, в том числе и автоматическая, является стандартным процессом при использовании веб-приложений. Собственный почтовый сервер создать легко, только необходимо указать адрес почтового сервера, который будет использован в качестве транспорта.

При построении простейшего почтового сервера требуются интерфейсы API JavaMail. Библиотека JavaMail содержит классы, позволяющие моделировать систему электронной почты. Класс **javax.mail.Session** представляет сеанс почтовой связи, класс **javax.mail.Message** — почтовое сообщение, класс **javax. mail.internet.InternetAddress** — адрес электронной почты.

Необходимую библиотеку, содержащую, в частности, архив **mail.jar**, следует загрузить по адресу oracle.com/technetwork/java/javamail/. Далее следует добавить этот файл в каталог jar-файлов серверу приложений (**lib** для Tomcat).

В случае автономного запуска без использования промежуточного сервиса для успешной работы почтового приложения бывает необходимо запустить почтовую программу James, являющуюся также одним из проектов **apache.org**.

Ниже приведена страница JSP, содержащая форму для заполнения основных полей: «**Кому**» — «**Send to**», «**Тема сообщения**» — «**Subject**», «**Текст письма**» — «**Body**»

 # 7 # страница создания электронного письма # index.jsp

```
<%@ page language="java" contentType= "text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
  <head>
    <title>JavaMail</title>
  </head>
  <body>
    <form action="email-action" method="POST">
      <table>
        <tr>
          <td>Send to:</td>
          <td>
            <input type="text" name="to" />
          </td>
        </tr>
        <tr>
          <td>Subject:</td>
          <td>
            <input type="text" name="subject" />
          </td>
        </tr>
      </table>
      <hr />
      <textarea type="text" name="body" rows="5" cols="45">
                                        Message text</textarea>
      <br />
      <br />
      <input type="submit" value="Send message!" />
    </form>
  </body>
</html>
```

Рисунок 6 - *Формирование запроса на отправку письма*

Параллельные процессы по отправке письма и предложению пользователю в это же самое время создать новое письмо организуются с применением потока в следующем сервлете.

```
// # 8 # сервлет почтового приложения # MailServlet.java
package by.bsu.mail.servlet;
import java.io.IOException;
import java.util.Properties;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import by.bsu.mail.action.MailThread;
import jakarta.servlet.annotation.WebServlet;

@WebServlet("/email-action")/// with init params
public class MailServlet extends HttpServlet {
   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse
response)
                                        throws ServletException, IOException {

  Properties properties = new Properties();
  ServletContext context = getServletContext();
  String filename = context.getInitParameter("mail");
  // запрос параметров почтового сервера из mail.properties
  // загрузка параметров почтового сервера в объект свойств
```

```
        properties.load(context.getResourceAsStream(filename));
        MailThread mailOperator = new MailThread(request.getParameter("to"),
                                        request.getParameter("subject"),
                                         request.getParameter("body"), properties);
        // запуск процесса отправки письма в отдельном потоке
        mailOperator.start();
        // переход на страницу с предложением о создании нового письма
        request.getRequestDispatcher("/send.jsp").forward(request, response);
          }
      }
```

Конфигурация почтового сервера размещена в файле **/config/mail.properties**.

```
      ########################################
      #########  Session  properties  #########
########################################
mail.smtp.host=smtp.gmail.com      mail.smtp.port=465
mail.user.name=REAL~MAIL~NAME@gmail.com
mail.user.password=REAL~PASSWORD
```

```
      <html>
```

```
      <html>
  <head>
    <title>Send Result</title>
  </head>
  <body>
    <p>Sending in progress...</p>
    <a href="index.jsp">Return to new mail page</a>
  </body>
</html>
```

В результате в браузер будет выведено:

Рисунок 7 - *Формирование запроса на отправку письма*

Здесь **Return to new mail page** представляет собой активную ссылку, перенаправляющую при запуске на **index.jsp** для создания еще одного письма. Процесс же отправки письма будет функционировать независимо от дальнейшей работы приложения.

Недостатком такого применения многопоточности будет отсутствие информации об успешном отправлении письма или возникших при этом ошибках и исключениях. В данной ситуации такую информацию можно только сохранить в логах или выдавать на специальных страницах. Поэтому современные почтовые онлайн системы и не используют многопоточность в таком виде, так как пришлось бы делать достаточно сложный интерфейс по сообщению пользователю об ошибках при отправке почты, что сделало бы работу в системе более сложной и противоречивой.

// # 10 # поток, отправляющий почтовые сообщения # MailThread.java

```
package by.bsu.mail.action;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import by.bsu.mail.creator.SessionCreator;

public class MailThread extends Thread {
   private MimeMessage message;
   private String sendToEmail;
   private String mailSubject;
   private String mailText;
   private Properties properties;
   public MailThread(String sendToEmail, String mailSubject, String mailText,
                                                    Properties properties) {
  this.sendToEmail = sendToEmail;
  this.mailSubject = mailSubject;
  this.mailText = mailText;
  this.properties = properties;
   }
   private void init() {
         // объект почтовой сессии
```

```
        Session mailSession = (new SessionCreator(properties)).createSession();
    mailSession.setDebug(true);
    // создание объекта почтового сообщения
    message = new MimeMessage(mailSession);
    try {
       // загрузка параметров в объект почтового сообщения
       message.setSubject(mailSubject);
       message.setContent(mailText, "text/html");
              message.setRecipient(Message.RecipientType.TO, new
                                                    InternetAddress(sendToEmail));
    } catch (AddressException e) {
       System.err.print("Некорректный адрес:" + sendToEmail + "  " + e);//log
    } catch (MessagingException e) {
       System.err.print("Ошибка формирования сообщения" + e);//log
    }
     }
     public void run() {
    init();
    try {
       // send mail
       Transport.send(message);
    }
        catch (MessagingException e) {
       System.err.print("Ошибка при отправлении сообщения" + e);//log
    }
     }
}
```

*// # 11 # конфигурирование почтовой сессии # SessionCreator.java */*

```
package by.bsu.mail.creator;
import java.util.Properties;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;

public class SessionCreator {
   private String smtpHost;
   private String smtpPort;
   private String userName;
   private String userPassword;
```

```
  private Properties sessionProperties;
  public SessionCreator(Properties configProperties) {
smtpHost = configProperties.getProperty("mail.smtp.host");
smtpPort = configProperties.getProperty("mail.smtp.port");
userName = configProperties.getProperty("mail.user.name");
userPassword = configProperties.getProperty("mail.user.password");
// загрузка параметров почтового сервера в свойства почтовой сессии
sessionProperties = new Properties();
sessionProperties.setProperty("mail.transport.protocol", "smtp");
sessionProperties.setProperty("mail.host", smtpHost);
sessionProperties.put("mail.smtp.auth", "true");
sessionProperties.put("mail.smtp.port", smtpPort);
sessionProperties.put("mail.smtp.socketFactory.port", smtpPort);
      sessionProperties.put("mail.smtp.socketFactory.class",
                            "javax.net.ssl.SSLSocketFactory");
sessionProperties.put("mail.smtp.socketFactory.fallback", "false");
sessionProperties.setProperty("mail.smtp.quitwait", "false");
  }
  public Session createSession() {
return Session.getDefaultInstance(sessionProperties,
                                  new javax.mail.Authenticator() {
   protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(userName, userPassword);
    }
 });
  }
}
```

Параметры контекста сервлета в файле **web.xml** для данного приложения представлены в виде:

```
    <context-param>
  <param-name>mail</param-name>
  <param-value>/WEB-INF/classes/config/mail.properties</param-value>
</context-param>
```

1.2. Java Server Page

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems (в настоящее время поглощена компанией Oracle), чтобы облегчить создание страниц с динамическим содержанием.

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения содержания и вида ответа, страницы JSP выполняют функцию отображения результатов работы приложения в виде текстовых документов типа HTML, XML, WML и некоторых других. JSP поддерживают как JavaScript, так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы.

Принято разделять динамическое и статическое содержимое JSP.

Динамические ресурсы. Результаты их деятельности изменяются во время выполнения приложения. Обычно представлены в виде выражений Expression Language, библиотек тегов и тегов разработчика.

Статические ресурсы. Не изменяются сами в процессе работы (HTML, JavaScript, изображения и т. д.).

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера в то время, как динамические нуждаются в движке (JSP Engine) и иногда в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: часть, состоящая только из динамических ресурсов, и часть, состоящая только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

— *разделение динамического и статического содержания.*

— *независимость от платформы*. Технологии Java не зависят от платформы, следовательно, JSP могут выполняться практически на любом веб-сервере. Разрабатывать JSP можно на любой платформе;

— *многократное использование компонентов.* Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоря-ет создание веб-приложений;

— *теги.* Спецификация JSP содержит библиотеку стандартных тегов JSTL, позволяет разработчику создавать собственные теги, кроме того, теги обеспечивают возможность использования JavaBean и обращение к классам биз-нес-логики.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответствен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят, как о thin-client (использование ресурсов клиента незначительно), потому что большая часть логики выполняется на сервере.

В спецификации JSP 1.2 были объявлены только пять основных тегов:

<%@ *директива* **%>** — используется для установки параметров серверной страницы JSP;

<%! *объявление* **%>** — (*нежелателен в современном программировании*) содержит поля и методы, которые вызываются в expression-блоке и становятся полями и методами генерируемого сервлета. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях;

<% *скриптлет* **%>** — (*нежелателен*) вживление java-кода в JSP-страницу. Скриптлеты обычно используют маленькие блоки кода и выполняются во вре-мя обработки запроса клиента. Когда все скриптлеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования. Контейнер помещает код Java в метод **_jspService(***paremeters***)** на этапе трансляции;

<%= *вычисляемое выражение* **%>** — (*нежелателен*) содержит операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**;

<%-- *JSP-комментарий* **--%>** — комментарий, который не отображается в исходных кодах JSP-страницы после этапа выполнения.

В конце прошлого тысячелетия существовала техника создания страницы на основе одного скриплета, в который вставлялся большой фрагмент кода, что в корне противоречило концепции JSP. В современных приложениях скрипле-ты, выражения и объявления не применяются вовсе из-за нарушения «теговой» структуры страницы.

1.2.1. Жизненный цикл

Процессы, выполняемые с файлом JSP при *первом* вызове: — браузер делает первый запрос к JSP;

— JSP-engine анализирует содержание файла JSP и создает сервлет с кодом, основанным на исходном тексте JSP, при этом engine транслирует статическое содержимое в методы вывода и помещает его в метод **_jspService()**. Полученный сервлет будет ответственен за генерацию статических элементов, определенных во время разработки. Динамические элементы трансли-руются в java-код;

— код сервлета компилируется в файл **.class**,

— создается объект;

— выполняется метод **init()** построенного сервлета. В итоге сервлет на основе JSP загружен в контейнер сервлетов и готов к работе;

— вызывается метод **_jspService()**, и jsp-сервлет логически исполняется, формируя экземпляр **response**;

— комбинация статического HTML и графики вместе с результатами исполне-ния динамических элементов, определенных в оригинале JSP, пересылают-ся браузеру через выходной поток объекта ответа **HttpServletResponse**.

— метод **destroy()** один раз при выгрузке

Следующие обращения к файлу JSP просто вызовут метод **_jspService()** сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен или сервлет не будет выгружен из контейнера. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности, в его метод **_jspService()**.

 # 2 # простейшая страница JSP # simple.jsp

```
<html>
  <head><title>Simple</title></head>
<body>
  Hello, Bender
  </body>
</html>
```

то в результате запуска в браузер будет выведено:

Hello, Bender

а код сервлета на базе JSP будет выглядеть следующим образом:

// # 3 # сгенерированный сервлет # simple_jsp.java

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
public final class simple_jsp extends org.apache.jasper.runtime.HttpJspBase
implements org.apache.jasper.runtime.JspSourceDependent {
  private static java.util.List _jspx_dependants;
  public Object getDependants() {
    return _jspx_dependants;
  }
  public void _jspService(HttpServletRequest request, HttpServletResponse response)
```

```
        throws java.io.IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
JspWriter _jspx_out = null;
PageContext _jspx_page_context = null;
try {
   _jspxFactory = JspFactory.getDefaultFactory();
   response.setContentType("text/html");
   pageContext = _jspxFactory.getPageContext(
                        this, request, response, null, true, 8192, true);
   _jspx_page_context = pageContext;
   application = pageContext.getServletContext();
   config = pageContext.getServletConfig();
   session = pageContext.getSession();
   out = pageContext.getOut();
   _jspx_out = out;

   out.write("<html><head>\r\n");
   out.write("<title>Simple</title>\r\n");
   out.write("</head>\r\n");
   out.write("<body>\r\n");
   out.write("Hello, Bender\r\n");
   out.write("</body></html>");
} catch (Throwable t) {
   if (!(t instanceof SkipPageException)) {
      out = _jspx_out;
      if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
      if (_jspx_page_context != null)
            _jspx_page_context.handlePageException(t);
   }
} finally {
   if (_jspxFactory != null)
```

```
                _jspxFactory.releasePageContext(_jspx_page_context);
            }
        }
    }
```

Следует обратить внимание на объявление в методе **_jspService()** целого ряда локальных переменных. Ко всем этим переменным возможен доступ пря-мо из текста JSP по имени в выражениях Expression Language.

1.2.2. Неявные объекты в expression language

JSP expression language определяет свое множество неявных объектов, пере-секающееся с множеством неявных объектов для JSP. Страница JSP с помо-щью скриптлетов всегда имеет доступ ко всем локальным переменным и пара-метрам метода **_jspService()** сервлета, создаваемым jsp-engine на основе этой же страницы. Наиболее очевидные:

— **pageContext** — определяет контекст JSP-страницы и предоставляет доступ к другим неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы;

— **servletContext —** определяет контекст для сервлета JSP страницы и любого веб-компонента, содержащегося в этом приложении;

— **param** — содержит все параметры, переданные странице в виде параметров формы или параметров адресной строки;

— **paramValues** — содержит список всех значений параметров, переданных на страницу;

— **initParam** — содержит конфигурационные параметры страницы, заданные в файле **web.xml**;

— **cookie** — содержит список переменных, переданных с файлом cookie;

— **request** — представляет запрос клиента. Обычно объект является экземпля-ром класса, реализующего интерфейс **javax.servlet.http.HttpServletRequest**. Для протокола, отличного от HTTP, это будет объект реализации интерфей-са **javax.servlet.ServletRequest**.

— **response** — представляет собой ответ клиенту. Обычно объект является экзем-пляром класса, реализующего интерфейс **javax.servlet.http.HttpServletResponse**. Для протокола, отличного от HTTP, это будет объект реализации интерфей-са **javax.servlet.ServletResponse**. Область видимости в пределах страницы;

— **config** — содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**. Область видимости в пределах страницы;
— **session** — создается контейнером в соответствии с протоколом HTTP и является экземпляром класса **javax.servlet.http.HttpSession**, предоставляет информацию о сессии клиента, если такая была создана. Область видимо-сти в пределах сессии;
— **out** — содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту. Объект является экземпляром класса **javax.servlet.jsp.JspWriter**. Область видимости в пределах страницы;
— **page** — явная ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**. Область видимости в пределах страницы;
— **exception** — представляет исключение одного из подклассов **java.lang.Throwable**, которое передается странице, помеченной как error page. Область видимо-сти в пределах страницы ошибок.

К неявным объектам возможно и обращение по имени, не допускающее никаких сокращений.

Существуют также объекты, которые обеспечивают доступ к различным областям видимости, а именно: **pageScope**, **requestScope**, **sessionScope**, **applicationScope**.

 # 4 # неявные объекты в EL # el_instance.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>EL for pageContext</title></head>
<body>
Путь к контексту : ${ pageContext.request.contextPath }
<br/> Имя хоста : ${ header["host"] }
<br/>Тип и кодировка контента : ${pageContext.response.contentType}
<br/> Кодировка ответа : ${pageContext.response.characterEncoding}
<br/> ID сессии : ${pageContext.request.session.id}
<br/>Время создания сессии в мсек :
${pageContext.request.session.creationTime}
<br/>
Время последнего доступа к сессии :
${pageContext.request.session.lastAccessedTime}
<br/> Имя сервлета : ${pageContext.servletConfig.servletName}
</body>
</html>
```

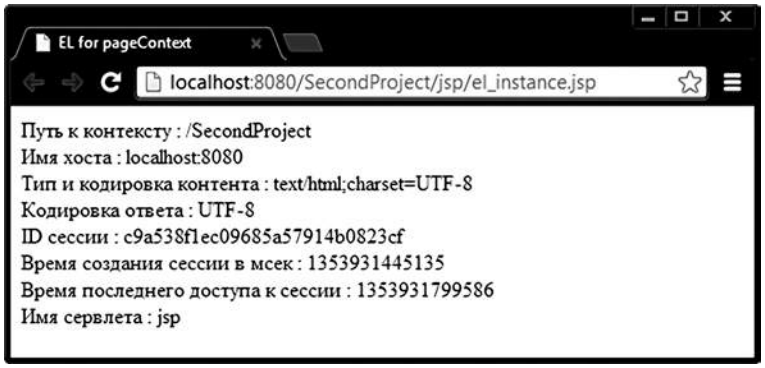


Рисунок 8 - *Обращение к неявным объектам*

1.2.3. Стандартные элементы action

Большинство стандартных тегов, содержащих символ %, в современном программировании не применяются. Используются action-теги версии JSP 2.0. Они позволяют создавать правильные JSP — документы. В тоже время декла-рации, выражения и скриптлеты остаются под негласным запретом из-за явно-го внедрения java-кода в JSP.

Action-теги:

— **<jsp:include>** — позволяет включать статические и динамические ресурсы в контекст генерируемой страницы при запросе вида

<jsp:include page="*относительный URL*" flush="true"/>

Включаемая страница не может объявлять собственные заголовки, опреде-ляющие тип или кодировку;

— **<jsp:declaration>** — (*нежелателен*) объявление, аналогичен тегу **<%! %>**;

— **<jsp:scriptlet>** — (*нежелателен*) скриптлет, аналогичен тегу **<% %>**;

— **<jsp:expression>** — (*нежелателен*) выражение, аналогичен тегу **<%= %>**;

— **<jsp:text>** — вывод текста;

— **<jsp:useBean>** — объявление экземпляра компонента Java Bean или класса, обладающего **public**-конструктором по умолчанию. Если экземпляр с указанным идентификатором не существует, то он будет создан в области видимости **scope** со значением **page** (страница), **request** (запрос), **session** (сессия) или **application** (приложение). Объявляется, как правило, с атрибутами **id** (имя объекта), **class** (полное имя класса), **type** (по умолчанию **class**).

```
<jsp:useBean id="ob" scope="request" class="by.bsu.test.Message" />
```

что идентично java-коду:

```
by.bsu.test.Message ob = new by.bsu.test.Message();
```

Создан объект **ob** класса **Message**, и в дальнейшем через этот объект можно вызывать доступные методы класса. По умолчанию область видимости устанавливается в значение **page**. Специфика компонентов JavaBean в том, что если компонент имеет поле **text**, экземпляр компонента имеет параметр **text**, а метод, устанавливающий значение, должен называться **setText(String txt)**, возвращающий значение — **getText()**.

// # 5 # простой Bean-class # Message.java

```
package by.bsu.test;
public class Message implements Serializable {
   private Integer id;
   private String text = "";
   public Message() {
   }
   public Message(Integer id, String text) {
      this.id = id;
      this.text = text;
   }
   public void setText(String text) {
      this.text = text;
   }
   public String getText() {
      return text;
   }
   public Integer getId() {
      return id;
   }
   public void setId(Integer id) {
      this.id = id;
   }
   @Override
   public String toString() {
      return "Message [id=" + id + ", text=" + text + "]";
   }
}
```

В теге **useBean** можно создавать только объекты классов, у которых определен конструктор по умолчанию;

— **<jsp:setProperty>** — позволяет устанавливать значения полей указанного в атрибуте **text** объекта. Если установить значение **property** в «*», то значе-ния свойств компонента JavaBean будут установлены таким образом, что будет определено соответствие между именами параметров и именами ме-тодов-установщиков (setter-ов) компонента:

```
<jsp:setProperty name="ob" property="text" value="hello" />
```

что идентично java-коду:

```
ob.setText("hello");
```

или с автоматическим преобразованием строки в число

```
<jsp:setProperty name="ob" property="id" value="717" />
```

идентично:

```
ob.setId(Integer.parseInt("717"));
```

что возможно только для интегральных и логических (boolean) типов;

— **<jsp:getProperty>** — извлекает значения поля указанного объекта, преобра-зует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="text" />
```

— **<jsp:forward>** — позволяет передать запрос другой странице или сервлету:

```
<jsp:forward page="относительный URL"/>
```

— **<jsp:params>** — группирует параметры внутри тега **jsp:plugin**;

— **<jsp:param>** — добавляет параметры в объект запроса, например, в эле-ментах **forward**, **include**, **plugin**;

— **<jsp:fallback>** — указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.

В качестве примера можно привести следующий фрагмент:

Элементы **<jsp:attribute>**, **<jsp:body>**, **<jsp:invoke>**, **<jsp:doBody>**, **<jsp:element>**, **<jsp:output>** используются, в основном, при включении в стра-ницу пользовательских тегов и библиотек тегов.

Современные тенденции оформления клиентской части стремятся к XML-формату, например, в виде **xhtml**. Такой формат менее подвержен грамматиче-ским ошибкам разработчика, так как осуществляется проверка на well-formed. Предпочтительно создавать JSP-страницу в виде JSP-документа — коррект-ного XML-документа, который ссылается на определенное пространство имен, содержит стандартные действия JSP, пользовательские теги и теги ядра JSTL, XML-эквиваленты директив JSP. JSP-документы сохраняются с расширением **.jspx**. Директива **page** для обычной JSP:

```
<%@ page contentType="text/html"%>
```

для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Если к директиве **page** добавить атрибут **session="false"**, то атрибуты сес-сии будут недоступны.

Директива **include**:

<%@ include file="header.jsp"%>

Action-тег:

<jsp:include file="header.jsp" />

Директива **taglib** для обычной JSP:

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

для JSP-документа:

<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>

Выше были представлены директива и action-тег **include**. При включении статических ресурсов в страницу оба механизма работают идентично. При включении динамических ресурсов (другой JSP или JSPF-фрагмента) директива **include** сразу конвертирует их непосредственно в создаваемый сервлет и включает в процесс исполнения, таким образом, исходная и включаемая страница могут пользоваться одним и тем же пространством имен. Применение action-тега **include** формирует вид запрашиваемой страницы уже после выполнения сгенерированного сервлета и статически включает в его объект **response**.

Таким образом, для включения динамического содержимого в JSP-страницу каждый раз, когда та получает запрос, используется директива **include**. В этом случае включаемые сегменты имеют доступ к объектам **page**, **request**, **session** и **application** исходной страницы и ко всем атрибутам, которые имеют эти

объекты. Если использовать action-тег **jsp:include**, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу).

Для иллюстрации отличий директивы и action-тега **include** следует объявить некоторую jsp с обращением к переменной с именем **calendar**, которая на данной странице не объявлялась, а именно:

 # 6 # страница для вставки в другую страницу # time.j

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
  <body>
    <hr/>
      Время в миллисекундах: ${calendar.timeInMillis}
    <hr/>
  </body>
```

```
</html>
```

Теперь в **index.jsp** будут использованы и директива, и action-тег.

 # 7 # страница, в которую вставляется контент другой

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
  <head><title>Index</title></head>
  <body>
    <jsp:useBean id="calendar" scope="page" class="java.util.GregorianCalendar"/>
    Directive
    <%@ include file="time.jsp"%>
    <br/>
    Action-tag
    <jsp:include page="time.jsp"/>
  </body>
</html>
```

В результате имеется: если страница вставлена с помощью директивы **include**, то ее объявления получают доступ к переменным в области видимости основной страницы.

Другой способ импорта статического содержимого представляет тег **<c:import>** из библиотеки Java Standard Tag Library (JSTL).

Защита от кнопки back браузера (в фильтр):

```
httpResponse.setHeader("Cache-Control","no-cache, no-store, must-revalidate");
response.addHeader("Cache-Control", "post-check=0, pre-check=0");
httpResponse.setHeader("Pragma","no-cache");
httpResponse.setDateHeader ("Expires", 0);
```

1.2.4. Expression Language

В JSP API введено понятие Expression Language (EL). EL используется для упрощения доступа к данным (атрибутам, параметрам и т. п.), хранящимся в различных областях видимости: **page**, **request**, **session, application** и вычис-ления простых выражений.

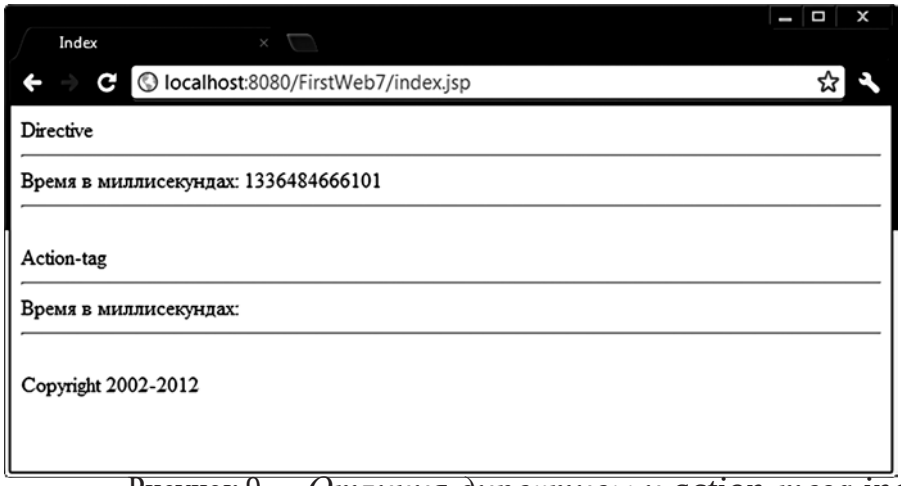


Рисунок 9 - *Отличия директивы и action-тега include*

EL вызывается при помощи конструкции **${***attribute_name***}**.

Начиная с версии спецификации JSP 2.0, EL является частью JSP и поддерживается без всяких сторонних библиотек.

EL-идентификатор ссылается на переменную, возвращаемую вызовом вида **pageContext.findAttribute(***attribute_name***)**. В общем случае переменная мо-жет быть сохранена в любой области видимости: **page** (**PageContext**), **request** (**HttpServletRequest**), **session** (**HttpSession**), **application** (**ServletContext**). Перечислены в порядке возрастания длительности существования. Если переменная не найдена, возвращается **null**. Значение **null** в поток вывода не пере-дается. Также возможен доступ к параметрам запроса через предопределенные объекты **param** и **paramValues**, а также к заголовкам запроса через **requestHeaders** и некоторым другим, как уже было сказано.

Данные, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой контейнеры — такие как, **List**, **Set**, **Map**, массивы и др. EL предоставляет доступ к этим объектам при помощи операторов **«.»** и **«[]»**. Способ применения этих операторов зависит от типа объекта. К элементу массива или списка доступ производится обычной индексацией с оператором **«[]»**. К элементам типа **Map**: либо указанием ключа после опера-тора «точка», либо обращением к ключу также по имени, но в операторе **«[]»**.

Обращение к экземпляру класса производится с неявным вызовом метода **toString()** в виде:

```
${ ob }
```

Обращение к значению поля экземпляра класса производится с неявным вызовом метода **get***ИмяПоля***()** в виде:

${ ob.*имяПоля* }

то есть на объекте будет вызван метод **get***Имя***()**.

 # 8 # вызов метода через bean-экземпляр # el_object.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
    <head><title>EL for object</title></head>
    <body>
        <jsp:useBean id="ob" scope="page" class="by.bsu.test.Message" />
        <jsp:setProperty name="ob" property="text" value="Bender" />
        ${ ob.text } in Rio
    </body>
</html>
```

С помощью оператора «точка» можно вызывать другие методы класса, к которому принадлежит объект. В этом случае сигнатура метода должна быть представлена полностью без каких-либо сокращений.

Если JSP вызывается из сервлета, передача объекта в качестве значения атрибута запроса осуществляется в виде:

```
Message obj = new Message();
obj.setText("Bender"); request.setAttribute("ob",
obj);
```

В результате запуска этой страницы в браузер будет выведено:

1.2.5. Bender in Rio

Создать объект-список и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

 # 9 # доступ к элементам списка # el_list.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
    <head><title>List</title></head>
    <body>
        Элемент списка: ${ lst[0] } <br/>
        Значение поля элемента списка: ${ lst[0].text } <br/>
```

```
            Весь список: ${ lst }
        </body>
    </html>
```

При вызове **el_list.jsp** из сервлета, что значительно более естественно, чем создание объекта в самой странице, передача объекта осуществляется с помо-щью экземпляра запроса в виде:

```
MessageList<Message> list = new MessageList<>();
request.setAttribute("lst", list);
```

где класс **MessageList** представляет собой реализацию списка в виде наследования или агрегирования (объявление в качестве поля класса):

// # 10 # список, наследующий коллекцию # Message

```
package by.bsu.test;
import java.util.ArrayList;
public class MessageList extends ArrayList<Message> {
   {
        /* логический блок вставлен для упрощения создания коллекции
        в реальном коде его быть не должно!*
      this.add(new Message(721, "Hello"));
      this.add(new Message(315, "Bye"));
   }
   public MessageList() {
   }
   @Override
   public String toString() {
      String str = "";
      for (Message msg : this) {
         str += msg;
      }
      return str;
   }
}
```

В браузер будет выведено:

Рисунок 10 - *Использование EL для доступа к элементам списка*

Создать объект-карту типа **Map** и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

 # 11 # доступ к элементам карты отображения # el

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Map</title></head>
  <body>
    Доступ по прямому значению ключа: ${ map.Blinov } <br/>
    Доступ по ключу: ${ map["Romanchik"] } <br/>
    Доступ к полю ключа: ${ map["Romanchik"].id } <br/>
    Вся карта: ${ map }
  </body>
</html>
```

Если **el_map.jsp** вызывается из сервлета, добавление объекта к запросу производится в виде:

MessageMap<String, Message> messMap = **new** MessageMap<String, Message> (); request.setAttribute("map", messMap);

где класс **MessageMap** представляет примитивнейшую реализацию карты отображения:

// # 12 # карта отображений, наследующая HashMap

```
package by.bsu.test;
import java.util.HashMap;

public class MessageMap extends HashMap<String, Message> {
  {
```

```
        // упрощение
        put("Blinov", new Message(1, "Work"));
        put("Romanchik", new Message(865, "Holiday"));
    }
}
```

```
Доступ по прямому значению ключа: Message [id=1, text=Work]
Доступ по ключу: Message [id=865, text=Holiday]
Доступ к полю ключа: 865
Вся карта: {Blinov=Message [id=1, text=Work], Romanchik=Message [id=865, text=Holiday]}
```

Рисунок 11 - *EL для доступа к значениям карты отображения*

Чтобы организовать обработку коллекции через цикл, а не посредством пря-мого индексирования, следует применить тег **<c:forEach>** из библиотеки JSTL, о нем будет рассказано в главе 18.

Параметры запроса, определенные на одной странице, доступны на другой странице при прямом переходе с первой страницы на вторую.

 # 13 # определение и передача параметров с запроса

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Parameters</title></head>
   <body><form action="el_param_browse.jsp">
        First Name: <input type="text" name="firstname"><br/>
        e-mail 1: <input type="text" name="mail"><br/>
        e-mail 2: <input type="text" name="mail"><br/>

        <input type="submit" name="submit" value="submit">
     </form>
   </body>
```

EL с помощью неявных объектов **param** и **paramValues** получает доступ к параметрам, определенным на вызывающей странице:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>EL for parameter</title></head>
```

```
    <body>
      The firstname is ${ param.firstname }<br/>
      e-mail 1 is ${ param.mail }<br/>
      e-mail 1 is ${ paramValues.mail[0] }<br/>
      e-mail 2 is ${ paramValues.mail[1] }
    </body>
</html>
```

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

1.2.6. EL операторы

Стандартные операторы отношения:

== (или **eq**), **!=** (или **ne**), **<** (или **lt**), **>** (или **gt**), **<=** (или **le**), **>=** (или **ge**). *Арифметические операторы*: **+**, **-**, *****, **/** (или **div**), **%** (или **mod**). *Логические операторы*: **&&** (или **and**), **||** (или **or**), **!** (или **not**).

Применение некоторых из операторов приведено в следующем коде, где для наглядности после знака равно приведен потенциальный результат, ожидаемый в окне браузера:

 # 15 # операции в EL # el_operation.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>EL operation</title></head>
  <body>
    ${ 1 > (7/3) } = false
    ${ 7.0 >= 5 } = true
    ${ 100.0 == 100.000000000000001 } = true
    ${ 'Z' < 'a' } = true
    ${ 'dog' gt 'doc' } = true
    ${ 7.0E3 + 1.4 } = 7001.4
    ${ 17 mod 7 } = 3
  </body>
</html>
```

Оператор **empty** используется для проверки значения переменной на **null**, или «пустое значение». Термин «пустое значение» зависит от типа проверяемого объекта. Это понятие включает нулевую длину для строки или нулевой размер для коллекции или массива.

 # 16 # проверка объекта на null и значения его

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>operator empty</title></head>
  <body>
    <jsp:useBean id="ob" scope="page"
class="by.bsu.test.Message" />
    Message is not valid: ${not empty ob and empty ob.text }
  </body>
</html>
```

В браузер будет выведено: **Message is not valid: true**

Если в JSP отсутствует объявление или доступ к объекту **ob** типа **Message**, то при попытке обращения **${ob}** в результате ничего выведено не будет. Поток вывода игнорирует значения типа **null**. Более того, при обращении к полю т.е get-методу несуществующего объекта вида **${ob.text}** генерации исключения не произойдет. Если же объект существует в какой-либо области видимости, а обращение происходит к несуществующему полю, то будет сгенерировано исключение.

1.2.7. Обработка ошибок

При выполнении веб-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Возникает вопрос об их обработке. Если исключение в java-коде не обрабатывается, то оно попадает в контейнер сервлетов и получает код 500 с генерацией сообщения вида «500 Internal Server Error». Возникает также при вызове сервлетом метода **sendError(500)** на объекте **HttpServletResponse**. Такие действия производятся, как правило, для исключений времени выполнения. Исключения, генерируемые веб-приложением и не перехватываемые фильтром, сервлетом или JSP. При отсутствии запрашиваемой страницы генерируется ошибка с кодом 404. При запрете досту-па — код 403. При отсутствии ответа сервера в течение определенного времени — код 504. При передаче слишком длинной строки запроса — код 414. Для обработки исключений в зависимости от типа в приложении могут использоваться обычные JSP-страницы, специальные JSP для обработки ошибок или HTML-страницы. Для настройки соответствия ошибок и обработчиков используется элемент **error-page** файла **web.xml**.

Например, для обработки ошибки по коду:

```
<error-page>
        <error-code>404</error-code>
  <location>/jsp/errors/error404.jsp</location>
</error-page>
```

или для обработки страницей ошибок конкретного типа необработанного ра-нее исключения

```
<error-page>
        <exception-type>java.lang.RuntimeException</exception-type>
  <location>/jsp/errors/error_runtime.jsp</location>
</error-page>
```

В элементе **error-code** указывается код ошибки, в элементе **exception-type** — тип исключения.

Чтобы страница распознавалась приложением как «error page», следует установить значение атрибута **isErrorPage** в **true**:

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

Ниже приведен пример с генерацией кода ошибки 500. При запуске страни-цы в коде соответствующего ей сервлета генерируется исключение **javax.el.PropertyNotFoundException**, и управление передается странице **error_runtime.jsp** в соответствии с тегом **<error-page>** в **web.xml**.

 # 17 # генерация ошибки # generation.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Генерация исключения</title></head>
  <body>
    <jsp:useBean id="ob" class="java.lang.String"/>
    ${ ob.toString }
  </body>
</html>
```

Генерация исключения произойдет из-за обращения к несуществующему свойству экземпляра класса **String**.

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме. Информация о коде и типе ошибки, а также о месте ее возникновения, извлекается из неявного объекта **pageContext**, точнее, из его поля **errorData**, являющегося экземпляром класса **javax.servlet.jsp.ErrorData**.

 # 18 # страница ошибок # error_runtime.jsp

```
<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8"
                                    pageEncoding="UTF-8"%>
<html><title>Error Page</title>
  <body>
    Request from ${pageContext.errorData.requestURI} is failed <br/>
    Servlet name: ${pageContext.errorData.servletName} <br/>
    Status code: ${pageContext.errorData.statusCode} <br/>
    Exception: ${pageContext.exception} <br/>
    Message from exception: ${pageContext.exception.message}
  </body>
</html>
```

В результате запуска страницы **index.jsp** в браузер будет выведена информация об ошибке в виде:

Servlet name: jsp
Status code: 500
Exception: javax.el.PropertyNotFoundException: Property 'tostring' not found on type java.lang.String
Message from exception: Property 'tostring' not found on type java.lang.String

Для указания конкретной error page, обрабатывающей ошибки, которые возникают при выполнении только этой страницы, можно использовать директиву **page** с атрибутом **errorPage**:

<%@ page errorPage="*path***" %>**, где **path** — это путь к странице-обработчику ошибок из обычной исполняемой страницы, например:

```
<%@ page errorPage="/errors/error_runtime.jsp" %>
```

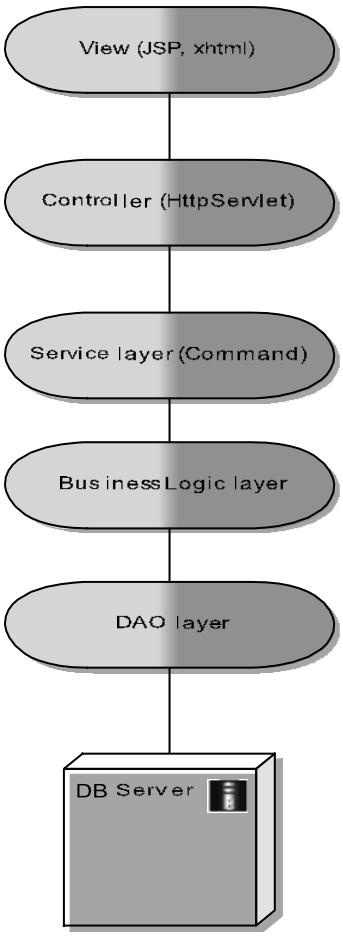


Рисунок 12 - *Слои веб-приложения*

1.2.8. Взаимодействие

JSP — сервлет — JSP

В большинстве приложений используются не сер-влеты или JSP по отдельности, а их архитектурное взаимодействие. Страница JSP представляет вид для результатов выполнения запроса клиента, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующую JSP и ее вызов. Т.е. сервлеты не генерируют от-вета сами, а только выступают в роли контроллера за-просов. Такая архитектура построения приложений носит название MVC. Model — классы бизнес-логики и длительного хранения (БД), View — страницы JSP, Controller — сервлет.

Применение шаблона на практике приводит к тому, что трехуровневая базовая модель распадается на мно-гоуровневую. Число и назначение слоев может существенно отличаться в зависимости от разработанного архитек турного решения.

В предлагаемом ниже подходе добавляются: уро-вень служб, обрабатывающих объект запроса, уровень логики, имплементирующий конкретные бизнес-процессы приложения и уровень организации взаимодей-ствия с базой данных.

Реализацию достаточно простой, но эффективной тех-нологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина и пароля пользователя с выводом приветствия в случае положительного результата. Вход в приложение осуществляется обычно через индексную страницу,

в данном случае **index.jsp**.

 # 19 # переход на страницу login # index.jsp

```
<%@ page contentType="text/html" %>
<html>
  <body>
    <jsp:forward page="/jsp/login.jsp"/>
  </body>
</html>
```

«Индексная» страница является формальной и выполняет **forward** запроса на первую активную страницу приложения. Следующая страница **login.jsp** содержит форму для ввода логина и пароля для аутентификации в системе:

 # 20 # форма ввода информации и вызов контроллера

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

```
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
<form method="POST" action="controller">
  <input type="hidden" name="command" value="login" />
  Login:<br/>
        <input type="text" name="login" value=""/>
        <br/>Password:<br/>
        <input type="password" name="password" value=""/> <br/>
        ${errorLoginPassMessage} <br/>
        ${wrongAction} <br/>
        ${nullPage} <br/>
        <input type="submit" value="Login"/>
      </form>
      <hr/>
  </body>
</html>
```

Страница содержит скрытое поле **hidden**, которое должно содержать указание на действие, связанное с функционалом этой страницы. В данном случае — запуск процесса аутентификации пользователя. В атрибут **errorLoginPassMessage** будет передаваться сообщение после неудачной попытки ввода логина-пароля. В атрибут **wrongAction** сообщение о несуществующей команде.

Демонстрационное приложение содержит страницу **main.jsp**, которая отображается пользователю в случае успешной аутентификации в приложении:

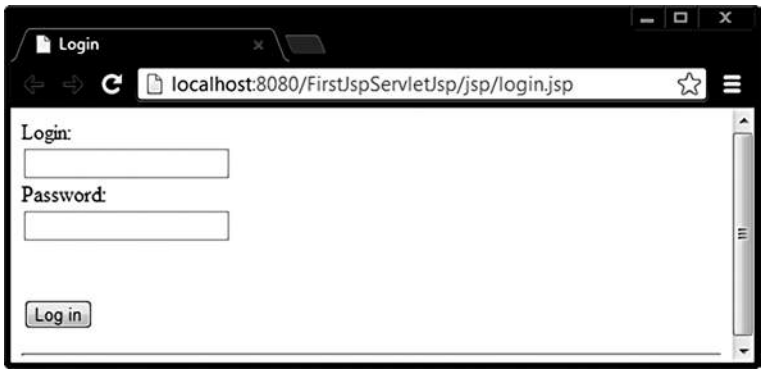


Рисунок 13 - *Страница аутентификации*

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <h3>Welcome</h3>
    <hr/>
      ${user}, hello!
    <hr/>
    <a href="controller?command=logout">Logout</a>
  </body>
</html>
```

Страница **main.jsp** также объявляет команду, для передачи параметра с командой здесь использована строка с запросом типа **GET**, но это лишь демонстрация возможностей. В реальном примере следует передавать ко-манду в скрытом поле и только по методу **POST**. Метод **doGet()** в сервлете оставить пустым.

Если пароль введен неправильно, то атрибут **errorLoginPassMessage** получит соответствующее значение и будет выведено:

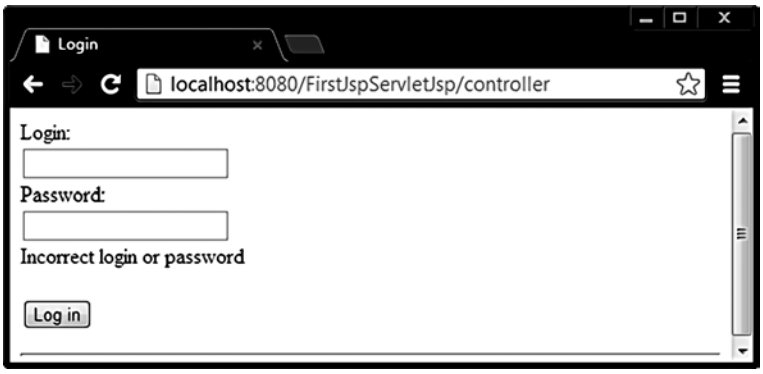


Рисунок 14 - *Вывод сообщения при ошибке аутентификации*

Страница **error.jsp**, последняя из созданных, загружается пользователю в случае возникновения ошибок:

 # 22 # страница вывода ошибок # /error/error.jsp

```
<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%> <html>
  <head>
    <title>Error Page</title>
  </head>
  <body>
    Request from ${pageContext.errorData.requestURI} is failed <br/>
    Servlet name or type: ${pageContext.errorData.servletName} <br/>
    Status code: ${pageContext.errorData.statusCode} <br/>
    Exception: ${pageContext.errorData.throwable}
  </body>
</html>
```

В эту страницу следует добавить возможность перехода на какую-либо область приложения. Такие страницы должны быть в любом приложении, в данном случае она также может содержать команду, позволяющую пользователю продолжить взаимодействие с системой.

Код сервлета-контроллера **Controller**, который не будет изменяться при добавлении новых страниц.

/ # 23 # контроллер запросов # Controller.java */*

```
package by.bsu.example.servlet;
@WebServlet("/controller")
public class Controller extends HttpServlet {
  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);
  }
  private void processRequest(HttpServletRequest request, HttpServletResponse response)
```

```
                    throws ServletException, IOException {
            String page = null;
            // определение команды, пришедшей из JSP
            Command command = ActionFactory.defineCommand(request);
            /*
             * вызов реализованного метода execute() и передача параметров
             * классу-обработчику конкретной команды
             */
            page = command.execute(request);
            // метод возвращает страницу ответа
            // page = null;
            // поэксперементировать!
            if (page != null) {
               RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(page);
               // вызов страницы ответа на запрос
               dispatcher.forward(request, response);
            } else {
               // установка страницы с сообщением об ошибке
               page = ConfigurationManager.getProperty("path.page.index");
               request.getSession().setAttribute("nullPage",
MessageManager.getProperty("message.nullpage"));
               response.sendRedirect(request.getContextPath() + page);
            }
         }
      }
```

Для определения типа и создания экземпляра-команды используется эле-мент реализации шаблона **Factory Method** класс **ActionFactory**.

На самом деле экземпляр **request** не должен свободно передаваться в бизнес-логику приложения во избежание некорректной модификации, поэтому следует предварительно извлекать необ-ходимую информацию из экземпляра-запроса и сохранять ее в объекте класса, специально для этого предназначенном, а именно:

```
/* # 24 # класс для хранения контента запроса # Sess
```

```
    public class SessionRequestContent {
  private HashMap<String, Object> requestAttributes;
  private HashMap<String, String[]> requestParameters;
  private HashMap<String, Object> sessionAttributes;
      private boolean invalidateSession;
      public SessionRequestContent(HttpServletRequest request) {
      // извлечение информации из запроса & session
      }
      public void restore(HttpServletRequest request) {
    // добавление в запрос & session данных для передачи в jsp
      }
    }
```

По окончании выполнения команд бизнес-логики, полученные данные следует передать в **request** для передачи клиенту. В такой ситуации метод **defineCommand()** в качестве параметра должен будет объявлять класс **SessionRequestContent**, а не **HttpServletRequest**.

При большом числе клиентов приложения решение определять все команды в перечислении не повлечет за собой создание большого количества объектов-команд, так как все клиенты будут пользоваться общими экземплярами команд.

/ # 26 # «хранилище» команд # CommandType.java */*

```
    package by.bsu.example.command.client;
import by.bsu.example.command.Command;
import by.bsu.example.command.LoginCommand;
import by.bsu.example.command.LogoutCommand;
public enum CommandType {
  LOGIN {
    {
      this.command = new LoginCommand();
    }
  },
```

```
        LOGOUT {
    {
        this.command = new LogoutCommand();
    }
  };
  Command command;
  public Command getCommand() {
    return command;
  }
}
```

```
    package by.epam.web.command;

    import java.util.Arrays;
    import java.util.Optional;

    public class CommandProvider {

        private CommandProvider() {
        }

        public static Optional<Command> defineCommand(String commandName)
{
            return Arrays.stream(CommandType.values())
                    .filter(command ->
command.getName().equalsIgnoreCase(commandName))
                    .findFirst()
                    .map(CommandType::getCommand);
        }
    }
```

Выполнение всех команд будет осуществляться при помощи простой реализации шаблона **Command**. Интерфейс **Command** определяет одно действие **execute()**, реализации интерфейса определяют в имплементированном ме-тоде бизнес-логику выполнения соответствующей команды.

```
    package by.bsu.example.command;
    import      javax.servlet.http.HttpServletRequest;
import by.bsu.example.logic.LoginLogic;
```

```
import by.bsu.example.resource.ConfigurationManager;
import by.bsu.example.resource.MessageManager; public
class LoginCommand implements Command {
    private static final String PARAM_NAME_LOGIN = "login";
    private static final String PARAM_NAME_PASSWORD = "password";
    private AbstractService service = new UserService();
    @Override
    public String execute(HttpServletRequest request) {
      String page = null;
      // извлечение из запроса логина и пароля
      String login = request.getParameter(PARAM_NAME_LOGIN);
      String pass = request.getParameter(PARAM_NAME_PASSWORD);
      if (service.checkLogin(login, pass)) {
        request.setAttribute("user", login);
        // определение пути к main.jsp
        page = ConfigurationManager.getProperty("path.page.main");
      } else {
        request.setAttribute("errorLoginPassMessage",
            MessageManager.getProperty("message.loginerror"));
        page = ConfigurationManager.getProperty("path.page.login");
      }
      return page;
    }
}
```

Ниже приведен код класса бизнес-логики **UserService**, который должен выполнять проверку правильности введенных логина и пароля с помощью запроса в БД, но для демонстрации простой логики работы приложения используется «заглушка» — логин и пароль хранятся в константах класса, чего никогда не следует делать в реальном проекте:

```
/* # 29 # псевдоаутентификация # UserReceiver.java
```

```
package by.bsu.example.logic;
public class UserService {
   private final static String ADMIN_LOGIN = "admin";
   private final static String ADMIN_PASS = "Qwe123";
   public boolean checkLogin(String enterLogin, String enterPass) {
      // валидация параметров логина и пароля
      //                    шифрование                    пароля
      return ADMIN_LOGIN.equals(enterLogin) && ADMIN_PASS.equals(enterPass);
   }
}
```

Команда **EmptyCommand** будет выполнена, если к сервлету произошло обращение без команды.

```
/* # 30 # пустая команда # EmptyCommand.java */

package by.bsu.example.command;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class EmptyCommand implements Command {// DefaultCommand
   @Override
   public String execute(HttpServletRequest request) {
      /* в случае ошибки или прямого обращения к контроллеру
            * переадресация на страницу     ввода логина */
      String page = ConfigurationManager.getProperty("path.page.login");
      return page;
   }
}
```

```
/*  #  31  #  команда  выхода  из  системы  #
```

```
package by.bsu.example.commands;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class LogoutCommand implements Command {
   @Override
   public String execute(HttpServletRequest request) {
      String page = ConfigurationManager.getProperty("path.page.index");
```

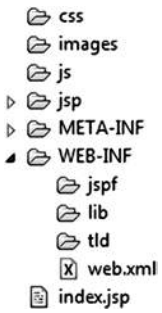
```
        // уничтожение сессии
        request.getSession().invalidate();
        return page;
    }
}
```

Служебные классы, извлекающие из properties-файлов необходимую для функ-ционирования приложения информацию:

```
/* # 32 # сообщения системы # MessageManager.java
package by.bsu.example.resource;
import java.util.ResourceBundle;
public class MessageManager {
    private final static ResourceBundle resourceBundle
=ResourceBundle.getBundle("resources.messages");
    // класс извлекает информацию из файла messages.properties
    private MessageManager() { }
    public static String getProperty(String key) {
        return resourceBundle.getString(key);
    }
package by.bsu.example.resource;
import java.util.ResourceBundle;
public class ConfigurationManager {
  private final static ResourceBundle resourceBundle =
ResourceBundle.getBundle("resources.config");
  // класс извлекает информацию из файла config.properties
    private ConfigurationManager() { }
    public static String getProperty(String key) {
        return resourceBundle.getString(key);
    }
}
```

Далее приведено содержимое файла *config.properties:*

```
##############################
## Application configuration ##
##############################
path.page.index=/index.jsp
path.page.login=/jsp/login.jsp
path.page.main=/jsp/main.jsp
path.page.error=/jsp/error/error.jsp
##############################
```



В общем случае генерация исключения в приложении и обращение по несу-ществующему адресу относятся к разл ичным типам, поэтому следует созда-вать отдельные error page. Данное приложение очень простое,

поэтому вторая страница ошибок явно лишняя.

Запуск приложения может производиться из командной строки браузера вида:

http://localhost:8080/FirstJspServletJsp**/index.jsp**

или просто

http://localhost:8080/FirstJspServletJsp/

Во втором случае также будет произведено обращение к **index.jsp**, так как этот адрес указан в элементе **<welcome-file-list>** де-скриптора приложения.

При большом количестве клиентов, работающих с приложением, к его базе данных выполняется большое количество запросов. Установление соединения с базой данных является дорогостоящей по требуемым ресурсам операцией. Эффективный способ решения данной проблемы — организация пула (pool) используемых соединений, которые не закрываются физически, а хранятся в очереди и предоставляются повторно для других запросов.

Пул соединений — это одна из стратегий предоставления соединений при-ложению (не единственная, да и самих стратегий организации пула существует несколько десятков).

Далее необходимо сконфигурировать параметры пула соединений для Apache Tomcat в файле **context.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/FirstJspServletJsp">
  <Resource
    name="jdbc/testphones"
    author="Container"
    type="javax.sql.DataSource"
    maxActive="32"
    maxIdle="8"
    maxWait="10000"
    username="root"
    password="pass"
    driverClassName="com.mysql.cj.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/testphones"
  />
</Context>
```

где:

name — имя ресурса, совпадающее по значению с именем, объявленным в **<res-ref-name>** дескриптора **web.xml**;

указание на тип интерфейса **type** также должно совпадать со значением в **<res-type>**;

username — имя владельца базы данных; **password** — пароль владельца; **driverClassName** — класс драйвера JDBC; **url** — адрес физического расположения БД;

maxActive — размер пула, то есть максимальное число активных соедине-ний, при превышении — ожидание освободившегося;

maxIdle — максимальное число свободных объектов-соединений; **maxWait** — время (миллисекунды) ожидания предоставления объекта-сое-

динения из пула соединений при невозможности немедленной выдачи соединения, например, если весь пул выбран. По истечении этого времени при не-предоставлении соединения генерируется исключение.

Непосредственно в коде приложения найти и запустить данный пул соединений можно с помощью JNDI (Java Naming Directory Interface).

Класс **javax.naming.InitialContext** как часть JNDI API обеспечивает взаимодействие с каталогом именованных объектов, а именно: хранение в некоторой

области необходимых классов объектов и предоставление к ним именованного доступа по запросу приложения (не только к БД, но и вообще к любому доступ-ному через JNDI). В каталоге ресурсов можно сопоставить объект источника данных **javax.sql.DataSource** с некоторым именем, предварительно создав объект **DataSource**.

Разделяемый доступ к источнику данных можно организовать, например, путем объявления статической переменной типа **DataSource** из пакета **javax.sql**, однако в JEE принято использовать для этих целей каталог или имя контекста приложения. Источник данных типа **DataSource** — это компонент, предостав-ляющий соединение с приложением СУБД.

Доступ к внешнему ресурсу предоставляется с помощью метода **lookup()** по его, ресурса, имени. Методу **lookup()** передается имя, всегда начинающееся с имени корневого контекста.

```
Context envCtx = (Context) (new InitialContext().lookup("java:comp/env"));
DataSource ds = (DataSource) envCtx.lookup("jdbc/testphones");
```

Соединение извлекается из пула очень просто:

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

После выполнения запроса соединение завершается, и его объект должен быть возвращен обратно в пул вызовом:

```
cn.close();
```

Производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса **DataSource**. Подобные классы есть и у MySQL и у Oracle. В этом случае пул соединений может быть создан с помощью класса **oracle.jdbc.pool.OracleDataSource**:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@server:1521/testphones ");
ods.setUser("root");
ods.setPassword("pass");
ods.setConnectionCacheName(CACHE_NAME); Properties
cacheProps = new Properties();
cacheProps.setProperty("MinLimit", "0");
cacheProps.setProperty("MaxLimit", "4");
cacheProps.setProperty("InitialLimit", "1");
```

```
        cacheProps.setProperty("ConnectionWaitTimeout", "5");
cacheProps.setProperty("ValidateConnection", "true");
ods.setConnectionProperties(cacheProps);
        Connection cn = ods.getConnection()
```

1.3. Сессии, события и фильтры

1.3.1. Сеанс (сессия)

При посещении клиентом веб-ресурса и выполнении вариантов запросов контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения информации о предыдущих действиях клиента. При этом возникают проблемы в распределенных системах с различными уровнями доступа для разных пользователей. Например, действия, которые может делать администратор системы, не может выполнять гость. В данном случае необходима проверка прав пользователя при переходе с одной страницы на другую. В иных случаях необходима информация о предыдущих запросах клиента. В JEE существует несколько способов хранения текущей информации о клиенте или о нескольких соединениях клиента с сервером.

Сеанс (сессия) между клиентом и сервером, устанавливаемая на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и веб-сервером в момент получения первого запроса к веб-приложению. Каждый клиент устанавливает с сервером свой собственный сеанс, который сохраняется до окончания работы с приложением.

Сеанс используется для обеспечения хранения данных при последователь-ном выполнении нескольких запросов различных веб-страниц или на обработку информации, введенной в пользовательскую форму в результате нескольких HTTP-соединений (например, клиент совершает несколько покупок в Интернет-магазине; студент отвечает на несколько тестов в системе дистанционного обучения). Как правило, при работе с сессией возникают следующие проблемы: — поддержка распределенной сессии (синхронизация/репликация данных, уни-

кальность идентификаторов и т. д.); — обеспечение безопасности;

— инвалидация сессии (expiration), предупреждение пользователя об уничтожении сессии и\или возможность ее продления (watchdog).

Чтобы открыть явный доступ к экземпляру сеанса/сессии пользователя веб-приложения, используются методы **getSession(boolean create)** или **getSession()** интерфейса **HttpServletRequest**. Экземпляр запроса возвращает ссылку на объект сессии. Метод не создает сессию, а только дает доступ

с помощью запроса к экземпляру сессии **HttpSession**, соответствующему данному пользователю.

Если для метода **getSession(boolean create)** входной параметр равен **true**, то сервлет-контейнер проверяет наличие активного сеанса, установленного с данным клиентом. В случае успеха метод возвращает дескриптор этого сеан-са. В противном случае метод устанавливает/создает новый сеанс:

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

после чего начинается сбор информации о клиенте.

Если метод **getSession(boolean param)** вызывать с параметром **false**, то ссылка на активный сеанс будет возвращена, если он существует, если же сеанс уничтожен или не был активирован, то метод возвратит **null**. Метод **getSession()** без параметров работает так же, как и **getSession(true)**.

Сессия содержит информацию о дате и времени создания последнего обращения к сессии, которая может быть извлечена с помощью методов **getCreationTime()** и **getLastAccessedTime()**.

Метод **String getId()** возвращает уникальный идентификатор JSEESSIONID (автоматически записывается в cookie), который получает каждый сеанс при создании. Метод **isNew()** возвращает **false** для уже существующего сеанса и **true** — для нового сеанса. Может быть изменен в процессы работы приложения. Задать время (в секундах) бездействия сессии, по истечении которого экземпляр сессии будет уничтожен, можно методом **setMaxInactiveInterval(long sec)**.

Чтобы сохранить значения переменной в текущем сеансе, используется ме-тод **setAttribute(String name, Object value)** класса **HttpSession**, прочесть — **getAttribute(String name)**, удалить — **removeAttribute(String name)**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя метод **Enumeration getAttributeNames()**, работающий так же, как и соответствующий метод интерфейса **HttpServletRequest**.

Принудительно завершить сеанс можно методом **invalidate()**. В результате для сеанса будут уничтожены все связи с используемыми объектами, и данные, сохраненные в старом сеансе, будут потеряны для клиента и всех приложений.

/ # 1 # добавление информации в сессию # SessionCo*

```
package by.bsu.control.listener;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
```

```
import java.io.IOException;
import jakarta.servlet.annotation.WebServlet;
@WebServlet(urlPatterns = {"/sessionservlet"})
public class SessionControlServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        if (session.getAttribute("role") == null) {
            session.setAttribute("role", "moderator");// guest
        }
        /* количество запросов, которые были сделаны
        к данному сервлету текущим пользователем
        в рамках текущей пользовательской сессии */
        Integer counter = (Integer) session.getAttribute("counter");
        if (counter == null) {
            session.setAttribute("counter", 1);
        } else {
        /* увеличивает счетчик обращений к текущему сервлету
        и кладет его в сессию */
            counter++;
            session.setAttribute("counter", counter);
        }
        request.setAttribute("lifecycle", "CONTROL request LIFECYCLE");
        request.getRequestDispatcher("/jsp/sessionattr.jsp").forward(request,
                                                    response);
        }
    }
```

Задача страницы **sessionattr.jsp** — отразить информацию о роли пользователя и счетчике обращений.

 # 2 # информация о сессии # /jsp/sessionattr.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html><head><title>Session Attribute</title></head>
<body>
    Role: ${role}
    <br /><hr />
    Counter: ${counter}
```

```
<br /><hr />
MaxInactiveInterval: ${pageContext.session.maxInactiveInterval}<br/>
ID session: ${pageContext.session.id}<br/>
Lifecycle: ${lifecycle}<br/>
<a href="index.jsp">Back to index.jsp</a>
</body></html>
```

В результате в браузер будет выведено:

В качестве данных сеанса выступают: счетчик обращений — объект типа **Integer** и роль пользователя. В ответ на пользовательский запрос сервлет **SessionServlet** возвращает страницу **sessionattr.jsp**, на которой отображаются все атрибуты сессии, а также идентификационный номер сессии и время инвалидации сессии. Время жизни сессии при отсутствии активности пользователя задано с помощью тега **session-config** в **web.xml** в виде:

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

где время ожидания задается в минутах.

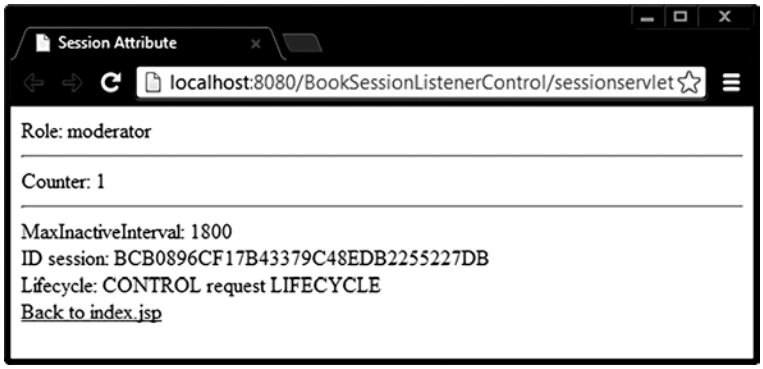


Рисунок 15 - *Информация о сессии*

 # 3 # стартовая страница # index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>Index Page</title></head> <body>
Lifecycle: ${lifecycle}
<a href="sessionservlet">Session Servlet</a>
</body></html>
```

В следующем примере произведено моделирование процесса ликвидации сессии при отсутствии активности за определенный промежуток времени. Реализация корректно работает только в случае, если приложением пользуется один клиент. Для нескольких клиентов алгоритм будем немного сложнее.

/ # 4 # инвалидация сессии по времени # TimeoutSer*

```
package by.bsu.timeoutsession;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
public class TimeoutServlet extends HttpServlet {
   protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                    throws ServletException, IOException {
      HttpSession session = null;
  if (SessionLocator.flag) {
            // "создание" сессии и установка времени инвалидации
     session = req.getSession();
     int timeLive = 10; // десять секунд!
     session.setMaxInactiveInterval(timeLive);
         SessionLocator.flag = false;
       } else {
      // если сессия не существует, то ссылка на нее не будет получена
      session = req.getSession(false);
      if (session == null) {
         SessionLocator.flag = true;
      }
   }
   req.setAttribute("messages", SessionLocator.addMessage(session));
       req.getRequestDispatcher("/jsp/time.jsp").forward(req, resp);
   }
}
```

/ # 5 # формирование сообщений для передачи в JS*

```
package by.bsu.timeoutsession;
import java.util.ArrayList;
import java.util.Date;
```

```
import jakarta.servlet.http.HttpSession;
public class SessionLocator {
   private final static String BR = "<br/><hr/>";
   public static boolean flag = true;
   public static ArrayList<String> addMessage(HttpSession session) {
      ArrayList<String> messages = new ArrayList<>();
      if (session != null) { // если сессия существует
         messages.add("Creation Time : " + new Date(session.getCreationTime()) +
BR);
         messages.add("Session id : " + session.getId() + BR);
         messages.add("Session alive!" + BR);
      } else {
          // если сессии уже не существует
          messages.add("Session disabled!" + BR);
      }
      return messages;
   }
}
```

При первом запуске в браузер будет выведено (см. рисунок 15).

Если повторить запрос к сервлету менее чем через 10 секунд, вывод будет идентично повторен. Если же запрос повторить более чем через десять секунд,сессия будет автоматически уничтожена, и в браузер будет выведено следую-щее сообщение:

Рисунок 16 - *Жизненный цикл сессии. Сессия «not alive»*

1.3.2. Файлы Cookie

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

Cookie — это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах cookies. Клиент может запретить браузеру прием файлов cookies. Браузер возвращает информацию обратно на сервер как часть заголовка HTTP, когда клиент выполняет переход на следующую страницу или повторно заходит на тот же веб-ресурс. Cookies могут быть ассоциированы не только с сервером, но и также с доменом; в этом случае браузер посылает их на все серверы указанного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (Single Signon), где серверы одного домена обмениваются идентификационными маркерами (token) с помощью общих cookies.

Cookie были созданы в компании Netscape как средства отладки, но теперь используются повсеместно. Файл cookie — это файл небольшого размера для хранения информации, который создается серверным приложением и размещается на компьютере пользователя. Браузеры накладывают ограничения на размер файла cookie и общее количество cookie, которые могут быть установлены на пользовательском компьютере приложениями одного веб-сервера.

Чтобы послать cookie клиенту, сервлет должен создать объект класса **Cookie**, указав конструктору имя и значение блока, и добавить их в объект-response. Конструктор использует имя блока в качестве первого параметра, а его значение — в качестве второго.

```
Cookie cookie = new Cookie("locale", "en_US");
response.addCookie(cookie);
```

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

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

После этого для каждого объекта класса **Cookie** можно вызвать метод **getValue()**, который возвращает строку **String** с содержимым блока cookie. В данном случае этот метод вернет значение «en_US».

Экземпляр **Cookie** имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из ключевых является срок жизни в секундах от момента первой отправки клиенту, определить который следует методом **setMaxAge(long sec)**. Если параметр не указан, то cookie существует только до момента прерывания контакта клиента с приложением.

В предложенном примере cookie добавляются в сервлете, в сервлете же инициируется процесс их извлечения и передачи информации, содержащейся в файле страницы **maincookie.jsp**.

 # 7 # информация из cookie # /jsp/maincookie.jsp

```
<%@ page contentType="text/html;charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>from Cookie </title></head>
<body>
   ${messages}
   <a href= "cookiecontroller">Messages from Cookie</a>
</body></html>
```

При старте приложения **${messages}** ничего не выведет, так как атрибут **messages** не существует. Сервлет **CookieController** добавляет cookie к объек-ту-ответу и пытается извлечь cookie из объекта-запроса. При первом запуске извлекать нечего, так как запрос пришел без cookie.

/ # 8 # создание и чтение cookie # CookieController*

```
package by.bsu.servlet.cookie;
import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import by.bsu.servlet.action;
public class CookieController extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse
response)
                                    throws ServletException, IOException {
      CookieAction.setCookie(response); // добавление cookie
      // извлечение cookie и добавление информации к request
      request.setAttribute("messages", CookieAction.addToRequest(request));
      request.getRequestDispatcher("/jsp/maincookie.jsp").forward(request,
                                                response);
   }
}
```

```
package by.bsu.servlet.action;
import java.util.ArrayList;
import java.util.Date;
import jakarta.servlet.http.HttpSession;
public class CookieAction {
   private static int number = 1;
   public static void setCookie(HttpServletResponse resp) {
      String name = "JamesBond";
  String role = "00" + number++;
  Cookie c = new Cookie(name, role);
  c.setMaxAge(60 * 60 * 24 * 30); // время жизни файла cookie
  resp.addCookie(c); // добавление cookie к объекту-ответу
  value = resp.getLocale().toString();
  Cookie loc = new Cookie("locale", value);

      resp.addCookie(loc);
   }
   public static ArrayList<String> addToRequest(HttpServletRequest request) {
      ArrayList<String> messages = new ArrayList<>();
  Cookie[ ] cookies = request.getCookies();
  if (cookies != null) {
     messages.add("Number cookies : " + cookies.length);
     for (int i = 0; i < cookies.length; i++) {
        Cookie c = cookies[i];
          messages.add(c.getName() + " = " + c.getValue());
          } // end for
  } // end if return messages;
   }
}
```

При первом запуске приложения cookie, естественно, переданы не будут, так как они еще не созданы, и браузер еще не получал ни одного ответа.

При повторном обращении к сервлету cookie уже будет передан с запросом и в браузер будет выведено:

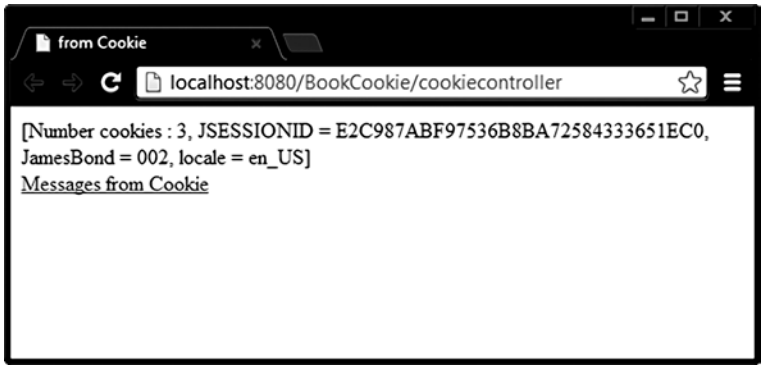


Рисунок 17 - *Информация из файла cookie*

Файл cookie, как это видно из указанного примера, может быть перезаписан при генерации каждого нового объекта-ответа.

1.3.3. Обработка событий

Изменение состояния некоторых объектов приложения представляют инте-рес и в чисто информационных целях, и в целях реализации реакции на проис-ходящее. Например, можно отслеживать время входа и выхода из системы пользователей, действия, выполняемые пользователем. Осуществлять инициализацию ресурсов в момент запуска системы, создания пользовательской сессии, и просто служить источником дополнительной информации.

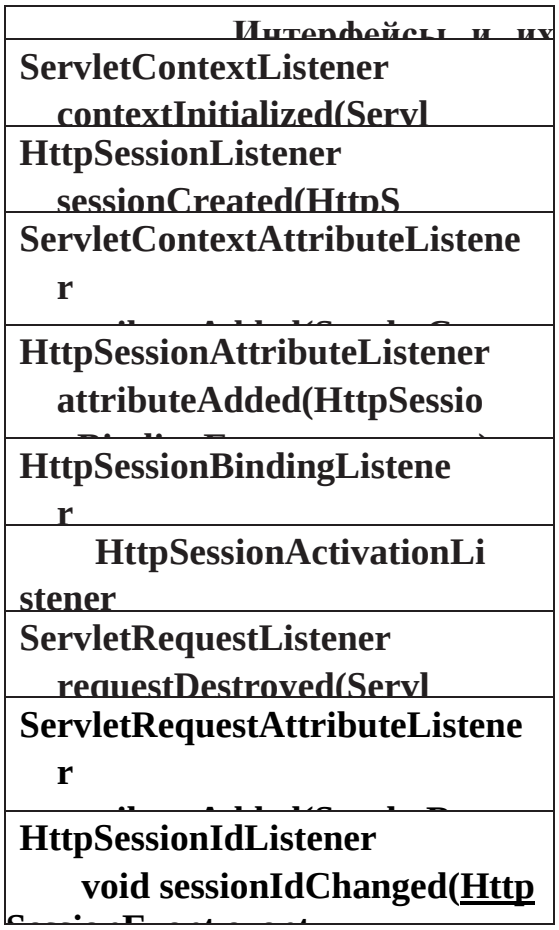


Рисунок 18 - *Интерфейсы событий и сигнатуры их методов*

Существует несколько интерфейсов, которые позволяют следить за событи-ями, связанными с сеансом, контекстом и запросом сервлета, генерируемыми во время жизненного цикла веб-приложения:

— **jakarta.servlet.ServletContextListener** — обрабатывает события создания/ удаления контекста сервлета;
— **jakarta.servlet.ServletContextAttributeListener** — обрабатывает события со-здания/удаления/модификации атрибутов контекста сервлета;
— **jakarta.servlet.http.HttpSessionListener** — обрабатывает события создания/ удаления HTTP-сессии;
— **jakarta.servlet.http.HttpSessionIdListener** — обрабатывает событие изменения идентификатора HTTP-сессии;
— **jakarta.servlet.http.HttpSessionAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов HTTP-сессии;

— **jakarta.servlet.http.HttpSessionBindingListener** — обрабатывает события привязывания/разъединения объекта с атрибутом HTTP-сессии;
— **jakarta.servlet.http.HttpSessionActivationListener** — обрабатывает события, связанные с активацией/деактивацией HTTP-сессии;
— **jakarta.servlet.ServletRequestListener** — обрабатывает события создания/удаления запроса;
— **jakarta.servlet.ServletRequestAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов запроса сервлета.

Регистрация блока прослушивания производится в дескрипторном файле приложения или аннотированием класса, реализующего интерфейс **Listener**.

Демонстрация обработки событий изменения состояния атрибутов сессии рассматривается на примере #1 данной главы. Обрабатываться будут события добавления атрибута **role**, а также добавления и изменения атрибута для счетчика обращений к сервлету **counter**. Для этого необходимо создать класс, реализующий интерфейс **HttpSessionAttributeListener** и реализовать необходимые для выполнения поставленной задачи методы.

```
/* # 10 # обработка событий добавления и изме
... # HttpSessionListenerImpl.java */
```

```
package by.bsu.control.listener;
import jakarta.servlet.annotation.WebListener;
import jakarta.servlet.http.HttpSessionAttributeListener;
import jakarta.servlet.http.HttpSessionBindingEvent;
@WebListener
public class HttpSessionListenerImpl implements HttpSessionAttributeListener {
   public void attributeRemoved(HttpSessionBindingEvent ev) {
   }
   public void attributeAdded(HttpSessionBindingEvent ev) {
      // запись в log-файл или иные действия
         System.out.println("add: " + ev.getClass().getSimpleName() + " : " +
                                 ev.getName() + " : " + ev.getValue());
   }
   public void attributeReplaced(HttpSessionBindingEvent ev) {
      // запись в log-файл или иные действия
      System.out.println("replace: " + ev.getClass().getSimpleName() + " : " +
                                 ev.getName() + " : " + ev.getValue());
   }
}
```

Экземпляр класса события **HttpSessionBindingEvent** дает доступ к самой сессии методом **getSession()**.

Зарегистрировать обработку событий можно также в элементе **<listener>** дескрипторного файла **web.xml**:

```
<listener>
  <listener-class>by.bsu.control.listener.SessionListenerImpl</listener-class>
</listener>
```

В результате запуска сервлета **SessionControlServlet** и двух обращений к нему в консоль будет выведено:

add: HttpSessionBindingEvent : role : moderator
add: HttpSessionBindingEvent : counter : 1
replace: HttpSessionBindingEvent : counter : 1
replace: HttpSessionBindingEvent : counter : 2

Интерфейс **ServletRequestListener** добавлен в Servlet API, начиная с версии 2.4. С его помощью можно отследить события создания запроса при обращении к сервлету и его уничтожения.

```
/* # 11 # обработка событий создания и уничто
```

```
package by.bsu.control.listener;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletRequestEvent;
import jakarta.servlet.ServletRequestListener;
import jakarta.servlet.annotation.WebListener;
import jakarta.servlet.http.HttpServletRequest;
@WebListener
public class ServletRequestListenerImpl implements ServletRequestListener {
   public void requestInitialized(ServletRequestEvent ev) {
   /* будет использован для получения информации о запросе
   HttpServletRequest req = (HttpServletRequest) ev.getServletRequest();
   String uri = "Request Initialized for " + req.getRequestURI();*/
      String id = "Request Initialized with ID=" + req.getRequestedSessionId();
      System.out.println(uri + "\n" + id);
      ServletContext context = ev.getServletContext(); // счетчик числа созданных
                                                       запросов
      Integer reqCount = (Integer) req.getSession().getAttribute("counter");
      if (reqCount == null) {
```

```
            reqCount = 0;
        }
        context.log(uri + "\n" + id + ", Request Counter =" + reqCount);
    }
    public void requestDestroyed(ServletRequestEvent ev) {
        System.out.println("Request Destroyed: "+
                           ev.getServletRequest().getAttribute("lifecycle"));
    }
}
```

С помощью экземпляра **ServletRequestEvent**, как видно из примера, можно получить доступ к экземпляру запроса **HttpServletRequest**, а через него и к сес-сии пользователя, и к контексту приложения.

После вызова страницы **index.jsp** будет создан новый **request** с нулевым значением атрибута **lifecycle:**

Request Initialized for /BookSessionListenerControl/index.jsp
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: null

При переходе к сервлету атрибут будет добавлен:

Request Initialized for /BookSessionListenerControl/sessionservlet
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: CONTROL request LIFECYCLE

После обратного перехода к **index.jsp** объект **request**, а вместе с ним атрибут **lifecycle**, будет уничтожен:

Request Initialized for /BookSessionListenerControl/index.jsp
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3 Request Destroyed: null

/ # 12 # обработка событий инициализации*

```
package by.bsu.control.listener;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;
@WebListener
public class ServletContextListenerImpl implements ServletContextListener {
```

```
  public void contextInitialized(ServletContextEvent ev) {
  ServletContext context = ev.getServletContext();
  String mailFeedback = context.getInitParameter("feedback");
  context.log("Context Initialized with parameter: " + mailFeedback);
  System.out.println("Context Initialized with parameter: " + mailFeedback);
   }
  public void contextDestroyed(ServletContextEvent ev) {
   }
}
```

Параметры инициализации представлены в **web.xml** в виде:

```
<context-param>
   <param-name>feedback</param-name>
   <param-value>blinov@gmail.com</param-value>
</context-param>
```

Так как событие инициализации контекста приложения производится только один раз за его жизненный цикл, то за все время работы приложения в log-файл только один раз будут выведены записи:

Dec 31, 2012 11:57:05 PM org.apache.catalina.core.ApplicationContext log INFO: Context Initialized with parameter: blinov@gmail.com

1.3.4. Фильтры

Реализация интерфейса **Filter** позволяет создать объект, который перехватывает запрос, может трансформировать заголовок и содержимое запроса кли-ента. Фильтры не создают запрос или ответ, а только модифицируют их. Фильтр выполняет предварительную обработку запроса, прежде чем тот попадает в сервлет, с последующей (если необходимо) обработкой ответа, исходящего из сервлета. Фильтр может взаимодействовать с разными типами ресурсов, в частности, и с сервлетами, и с JSP-страницами.

Основные действия, которые может выполнить фильтр:

— перехват инициализации\вызова сервлета или jsp и определение содержания запроса прежде, чем сервлет будет инициализирован\вызван;

— блокировка\перенаправление дальнейшего прохождения пары request-response;

— изменение заголовка и данных запроса и ответа;

— взаимодействие с внешними ресурсами; — построение цепочек фильтров;

— фильтрация более одного сервлета и/или jsp.

При программировании фильтров следует обратить внимание на интерфей-сы **Filter**, **FilterChain** и **FilterConfig** из пакета **jakarta.servlet**. Сам фильтр опре-деляется реализацией интерфейса **Filter**. Основным методом этого интерфейса является метод **doFilter(ServletRequest req, ServletResponse res, FilterChain chain)**, которому передаются объекты запроса, ответа и цепочки фильтров. Он вызывается каждый раз, когда запрос/ответ проходит через список фильтров **FilterChain**. В данный метод помещается реализация задач, обозначенных выше.

Кроме того необходимо реализовать метод **void init(FilterConfig config)**, который принимает параметры инициализации и настраивает конфигурацион-ный объект фильтра **FilterConfig**. Метод **void destroy()** вызывается при завершении работы фильтра, в тело которого помещаются команды освобождения используемых ресурсов.

Жизненный цикл фильтра начинается с однократного вызова метода **init()**, затем контейнер вызывает метод **doFilter()** столько раз, сколько запросов будет сделано непосредственно к данному фильтру. При отключении фильтра вызы-вается метод **destroy()**.

С помощью метода **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)** каждый фильтр получает текущий запрос и ответ, а также список фильтров **FilterChain**, предназначенных для обработки. Если в **FilterChain** не осталось необработанных фильтров, то продолжается переда-ча запроса/ответа. Затем фильтр вызывает **chain.doFilter()** для передачи управ-ления следующему фильтру.

Фильтр может модифицировать ответ сервера клиенту. Одним из распростра-ненных приемов использования фильтра является модификация кодировки запро-са и/или ответа. Когда страница посылает в сервлет запрос клиента, используется установленная в запросе кодировка. В настоящее время некоторыми браузе-рами кодировка устанавливается по умолчанию и не зависит от кодировки страницы, обращающейся к серверу, и к тому же вообще не передается сер-веру приложений. Для корректной интерпретации передаваемых с запросом, например, символов кириллицы, необходимо перехватить запрос и заменить его кодировку.

 # 13 # обращение к кодировке запроса, переданной

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Encoding Filter</title></head>
```

```
<body>
   Request encoding: ${ pageContext.request.characterEncoding }
</body></html>
```

Если фильтр не подключать, то кодировка запроса не будет определена, потому что браузер всегда передает значение **null** для кодировки запроса.

Реализация интерфейса **Filter** для поставленной задачи, изменяющей кодировку запроса и ответа на кодировку, заданную параметром фильтра.

```
// # 14 # фильтр, корректирующий кодир
```

```
package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;
@WebFilter(filterName = "Encoding", urlPatterns = { "/*" },
        initParams = { @WebInitParam(name = "encoding", value = "UTF-8",
                                     description = "Encoding Param") })
public class EncodingFilter implements Filter {
   private String code;
   public void init(FilterConfig fConfig) throws ServletException {
  code = fConfig.getInitParameter("encoding");
   }
   public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
  String codeRequest = request.getCharacterEncoding();
  if (codeRequest == null || !codeRequest.equalsIgnoreCase(code)) {
      request.setCharacterEncoding(code);
  }
  chain.doFilter(request, response);
   }
   public void destroy() {
  code = null;
   }
```

```
}
```

Параметры инициализации задаются внутри объявления аннотации **@WebFilter** аннотацией **@WebInitParam**. Аналогичная конфигурация для предыдущей версии определяется в дескрипторе **web.xml**:

```
<filter>
   <filter-name>encodingfilter</filter-name>
   <filter-class>by.bsu.sample.filter.EncodingFilter</filter-class>
   <init-param>
  <param-name>encoding</param-name>
  <param-value>UTF-8</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>encodingfilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Для использования фильтра для всех сервлетов и JSP используется шаблон "**/***", только для JSP — "***.jsp**", для каталога — "**/jsp/admin/***", для конкретной jsp — "**/index.jsp**", для сервлета — "**/controller**".

Фильтры применяются для обеспечения безопасности приложения, а имен-но, легко решают задачу запрещения несанкционированного прямого обраще-ния к JSP. Следствием установки шаблона **urlPatterns** в значение "**/jsp/***" будет вызов фильтра при любом прямом обращении из строки браузера. Фильтр в итоге перенаправит вызов на указанную страницу.

```
/* # 15 # фильтр, перенаправляющий все прямые об
```

```
package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;
```

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
@WebFilter( urlPatterns = { "/jsp/*" },
initParams = { @WebInitParam(name = "INDEX_PATH", value = "/index.jsp") })
public class PageRedirectSecurityFilter implements Filter {
   private String indexPath;
   public void init(FilterConfig fConfig) throws ServletException {
  indexPath = fConfig.getInitParameter("INDEX_PATH");
   }
   public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
  HttpServletRequest httpRequest = (HttpServletRequest) request;
  HttpServletResponse httpResponse = (HttpServletResponse) response;
  httpResponse.sendRedirect(httpRequest.getContextPath() + indexPath);
  chain.doFilter(request, response);
   }
   public void destroy() {
   }
}
```

Фильтр также успешно перехватывает обращения к сервлету. Например, обращение к сервлету из любого источника перехватывается следующим фильтром. Фильтр проверяет роль пользователя, извлекая его значение из сессии. Если роль не задана, то фильтр присваивает ему значение **GUEST** и перенаправляет клиента на страницу **guest.jsp**.

```
/* # 16 # фильтр, перенаправляющий всех новых п
```

```
package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
```

```
import jakarta.servlet.http.HttpSession;
import by.bsu.barrier.client.ClientType;
@WebFilter(urlPatterns = { "/controller" }, servletNames = { "MainServlet" })
public class ServletSecurityFilter implements Filter {
    public void destroy() {
    }
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
  HttpServletRequest req = (HttpServletRequest) request;
  HttpServletResponse resp = (HttpServletResponse) response;
  HttpSession session = req.getSession();

  ClientType type = (ClientType) session.getAttribute("userType");
  if (type == null || session == null) {
     type = ClientType.GUEST;
     session.setAttribute("userType", type);
     session.setAttribute("locale", "fr_CA");
     RequestDispatcher dispatcher = request.getServletContext()
                                    .getRequestDispatcher("/jsp/guest.jsp");
     dispatcher.forward(req, resp);
     return;
  }
  // pass the request along the filter chain
  chain.doFilter(request, response);
    }
    public void init(FilterConfig fConfig) throws ServletException {
    }
}
```

```
/* # 17 # перечисление с типами пользовате
```

```
package by.bsu.barrier.client;
public enum ClientType {
   GUEST, USER, ADMINISTRATOR
}
```

```
@WebFilter(urlPatterns = {"/*"})
public class CurrentPageFilter implements Filter {
   private static final String REFERER = "referer";
   private static final String PATH_REGEX = "/controller.+"
   public void destroy() {
```

```
    }
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpSession session = request.getSession(true);
        String url = request.getHeader(REFERER);
  //or  request.getRequestURI();
        String path = substringPathWithRegex(url);
        session.setAttribute("current_page", path);
        chain.doFilter(req, resp);
    }

    private String substringPathWithRegex(String url) {
        Pattern pattern = Pattern.compile(PATH_REGEX);
        String path = null;
        if (url != null) {
            Matcher matcher = pattern.matcher(url);
            if (matcher.find()) {
                path = matcher.group(0);
            } else {
                path = PagePath.PATH_PAGE_MAIN;
            }
        }
        return path;
    }

    public void init(FilterConfig config) throws ServletException {

    }
}
```

Фильтры в общем случае не перехватывают запросы, перенаправленные с помощью методов **forward()** и **include()** экземпляра **RequestDispatcher**. Элемент **dispatcherTypes** позволяет указать фильтру способ получить управление: непосредственно от сервлета или клиента значением **DispatcherType.REQUEST**, в результате перенаправления с использованием метода **forward()** значением **DispatcherType.FORWARD** или метода **include()** значением **DispatcherType.INCLUDE**) и некоторых других:

```
@WebFilter(dispatcherTypes = {
```

```
        DispatcherType.REQUEST,
        DispatcherType.FORWARD,
        DispatcherType.INCLUDE
    }, urlPatterns = { "/jsp/*" })
```

Конфигурировать доступ можно любым необходимым набором значений.

```
@WebFilter(dispatcherTypes = { DispatcherType.FORWARD }, urlPatterns = { "/jsp/admin/*" } )
```

1.4. JSP Standard Tag Library

Повторное использование кода — обычная техника программирования. В большинстве случаев такой код инкапсулируется в методе, при создании JSP — в стандартом теге. Наиболее стандартные и востребованные решения были определены и собраны в библиотеку JSTL.

JSP-страницы, включающие элементы action (стандартные действия) и пользовательские теги, не могут быть технологичными без использования JSTL (JSP Standard Tag Library). Создание страниц с применением JSTL позволяет упростить разработку и отказаться от вживления java-кода в JSP. Как было показано ранее, страницы со скриптлетами трудно читаемы, что вызывает проблемы как у программиста, так и у веб-дизайнера, не обладающего глубокими познаниями в Java.

Библиотека тегов JSTL состоит из пяти групп тегов: основные теги — **core**, теги форматирования — **formatting**, теги для работы с SQL — **sql**, теги для обработки XML — **xml**, функции-теги для обработки строк — **functions**.

JSTL предоставляет следующие возможности:

— поддержку Expression Language, что позволяет разработчику писать простые выражения внутри атрибутов тега и предоставляет «прозрачный» до-ступ к переменным в различных областях видимости страницы;

— организацию условных переходов и циклов, основанную на тегах, а не на скриптовом языке;

— простое формирование доступа (URL) к различным ресурсам;

— интернационализацию JSP;

— взаимодействие с базами данных (*не рекомендуется*); — обработку XML;

— форматирование и разбор строк посредством библиотеки функций.

Библиотека	Число	Описание
core	14	*Основные*: **if/then** выражения и конструкции множествен-ного выбора; вывод; создание и удаление контекстных пе-ременных; управление свойствами JavaBeans компонентов; перехват исключений; **forEach** для итерирования коллек-ций, массивов; создание URL и импорт их содержимого.
formatting	12	*Интернационализация и форматирование*: локализация текста и структуры сообщений; форматирование и анализ чисел, денежных единиц и дат.

Библиотека	Число тегов	Описание
sql	6	*Доступ к БД*: описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса.
xml	10	*XML-анализ и преобразование*: преобразование XML; до-ступ и преобразование XML через XPath и XSLT.
fn	16	*Строки*: обработка объектов типа String и определение размера коллекций.

Библиотеку JSTL версии 3.0.0 (или более позднюю версию) подключаются в виде

```
<dependency>
  <groupId>jakarta.servlet.jsp.jstl</groupId> <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
  <version>3.0.0</version>
</dependency>
```

Или

```
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>jakarta.servlet.jsp.jstl</artifactId> <version>3.0.1</version>
</dependency>
```

Expression Language подключается

```
<dependency>
  <groupId>jakarta.el</groupId>
  <artifactId>jakarta.el-api</artifactId>
  <version>5.0.1</version>
</dependency>
```

При указании значения параметра **xmlns** элемента **root** (для JSP-страницы значение параметра директивы **taglib uri="jakarta.tags.core"**) необходимо быть внимательным, так как при неправильном указании адреса JSP-страница не сможет получить доступ к тегам JSTL. Проверить корректность значения параметра **uri** (оно же справедливо и для параметра **xmlns**) можно в файле подключаемой библиотеки (например **c.tld**). Простейшая JSP с применением тега JSTL, выводящим в браузер приветствие, будет выглядеть следующим образом:

 # 1 # простой вывод # first.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:out value="Welcome to JSTL"/>
  </body>
</html>
```

Тег **<c:out/>** отправляет значение параметра **value** в поток **JspWriter**.

JSTL core

Библиотека **core** содержит в себе наиболее часто используемые решения.

Теги общего назначения:

<c:out /> — вычисляет и выводит значение выражения;

<c:set /> — создает и устанавливает переменную в указанную область видимости;

<c:remove /> — удаляет переменную из указанной области видимости;

<c:catch /> — перехватывает исключение.

Теги условного перехода:

<c:if /> — тело тега выполняется, если значение выражения **true**;

<c:choose /> (**<c:when />**, **<c:otherwise />**) — то же, что и **<c:if />** с поддержкой нескольких условий и действия, производимого по умолчанию.

Итераторы:

<c:forEach /> — выполняет тело тега для каждого элемента коллекции, массива, итерируемого объекта;

<c:forTokens /> — выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

<c:redirect/> — перенаправление на указанный адрес **URL**;

<c:import/> — добавляет на JSP содержимое указанного веб-ресурса;
<c:url/> — формирует адрес с учетом контекста приложения **request.getContextPath()**;
<c:param/> — добавляет параметр к запросу, сформированному при помощи **<c:url/>**.

Ниже приведено несколько примеров, иллюстрирующих применение основных тегов из группы **core**. Существует аналогия между тегом **<c:set>** и тегом **<jsp:useBean>**. Оба создают и помещают экземпляры в заданную область видимости. Но **<jsp:useBean>** только непосредственно создает экземпляр конкретного типа, а **<c:set>**, создав ссылку, позволяет извлекать значение, например, из параметров запроса, сессии и т. д.

 # 2 # демонстрация работы тегов c:set, c:if # jstl_set

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:set var="user" value="guest" scope="page"/>
    <c:if test="${ not empty user and user eq 'guest' }">
      User is Guest
    </c:if>
    <br/>
  </body>
</html>
```

В браузер будет выведено:

User is Guest

Тег **<c:if>** вычисляет значение атрибута **test**. Если результатом будет истина, как в приведенном выше примере, то тело тега выполняется, если ложь — игнорируется. Где понятию «ложь» соответствует любое значение, отличное от **true**.

Сохранить результаты вычисления в атрибуте **test** можно в другом атрибуте **var** с возможностью следующего использования в контексте страницы.

 # 3 # использование значения атрибута test # jstl_if

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
```

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core: if</title>
  </head>
 <body>
    <c:if test="${ 5 < 7 }" var = "firstOperation" scope = "page">
      first if : is TRUE
    </c:if>
    <br/>
    <c:out value="First Operation variable is : ${ firstOperation }"/>
    <br/>
    <c:if test="${ firstOperation } + some bt" var = "secondOperation">
      second if : is TRUE
    </c:if>
    <br/>
    <c:out value="Second Operation variable is : ${ secondOperation }"/>
  </body>
</html>
```

В результате в браузер будет выведено:

first if : is TRUE
First Operation variable is : true Second Operation variable is : false

Тег **<c:set>** может присваивать значение и без помощи атрибута **value**, про-сто объявляя его в теле тега:

```
<c:set var="user" scope="page">
  guest
</c:set>
```

Такая запись необходима в ситуации, когда размещение значения в атрибут **var** невозможно: например, при обязательном наличии в значении двойных или одинарных кавычек.

```
<c:set var="users" scope="page">
  "guest" 'client' "admin"
</c:set>
<c:out value="${users}"/>
```

Тег **<c:remove>** удаляет переменную из области видимости и при попытке доступа к ней после выполнения тега результатом будет **null**.

 # 4 # удаление атрибута # remove.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:set var="user" scope="page">
      "guest" 'client' "admin"
    </c:set>
```



```
    User is set= <c:out value="${user}" />
    <c:remove var="user"/>
    <br/>
    User after remove=<c:out value="${user}"/>
  </body>
</html>
```

После срабатывания тега **<c:remove>** атрибут **user** будет удален из области видимости и при попытке его поиска будет возвращено значение **null**, в браузе-ре не отображаемое.

1.4.1. Автоматическое приведение типов и перехват исключений

Данные не всегда имеют тот же тип, который ожидается в EL-операторе. EL использует набор правил для автоматического приведения типов. Например, если оператор ожидает параметр типа **Double**, то значение идентификатора будет приведено к типу **Double**.

 # 5 # приведение типов # type.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:set var="number" value="42.0" scope="session"/>
    <c:if test="${ number < 90.0 }">
      <c:out value ="Number ${ number }"/> is smaller than (90.0)
    </c:if>
```

```
    </body>
</html>
```

Параметр из переменной **number** передается в виде строки **String**, и преобразованное к числу значение будет корректным при использовании переменной. Причем совпадение по типу должно быть точным. Если ожидается тип **Long**, а передается строка в виде **Double**, приведение типов будет некорректным, что приведет к генерации исключения, как в предлагаемом коде

```
<c:set var="number" value="42.0" scope="session"/>
   <c:if test="${ number < 90 }">
   <c:out value ="Number ${ number }"/> is smaller than (90)
</c:if>
```

где число **42.0** не может быть приведено к ожидаемому типу **Long**. В итоге страница выполнена не будет и нормальная работа остановлена генерацией ошибки с кодом 500.

Для предотвращения такой ситуации и нормального завершения выполне-ния страницы следует использовать тег **<c:catch>**.

 # 6 # перехват исключения # jstl_catch.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
   <head>
      <title>Core: catch</title>
   </head>
      <body>
         <c:set var="number" value="42.0" scope="session" />
         <c:catch var="formatException">
            <c:if test="${ number < 90 }">
               <c:out value="Number ${ number }" /> is smaller than (90)
            </c:if>
         </c:catch>
         <br/>
         <c:if test="${not empty formatException }">
            Wrong format number: <c:out value="${ number }" /> isn't Long
            <br /><hr />
            Generated exception
            <hr />
```

```
            ${ formatException }
        </c:if>
    </body>
</html>
```

В итоге разработчик будет предупрежден о неправильных данных и сможет принять меры по недопущению некорректной информации в указанный параметр.

Тег **<c:catch>** введен для избирательной обработки одного или нескольких исключений, генерация которых не должна приводить к переходу на заданные страницы обработки ошибок. Страница, на которой произошло исключение, будет корректно исполнена до конца.

Выражение **<c:if test="${not empty formatException}">** позволяет определить, происходила ошибка или нет. В случае возникновения исключения переменная в атрибуте **var** тега **</c:catch>** инициализируется экземпляром **javax.el.ELException**, доступ к которому может быть получен в контексте страницы, как это и сделано в приведенном примере.

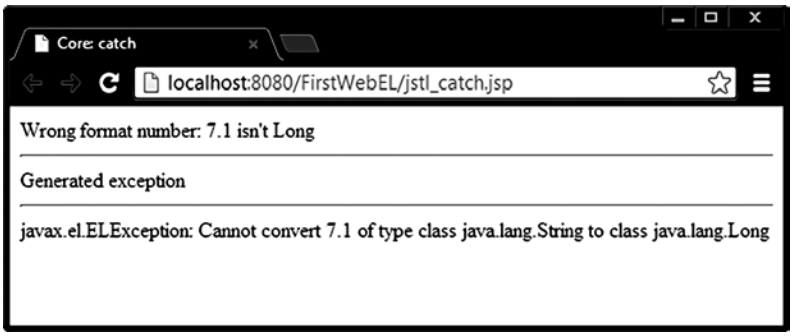


Рисунок 19 - *Генерация и перехват исключения*

1.4.2. Исключающие условия <c:choose>

Операции по управлению условным переходом существенно расширяются благодаря тегу **<c:choose>**. Выполнение производится только одного из всего набора действий тега после определения истинности условия. Все остальные выражения, даже истинные, игнорируются, после чего управление передается тегу, следующему после тега **</c:choose>**. Если ни одно из действий **<c:when>** не выполнено, то есть **test** не принял значение **true**, то управление переходит к единственному оператору **<c:otherwise>**, который может и отсутствовать.

 # 7 # множественный условный переход # choose

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```
  pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
   <head>
      <title>Core:choose</title>
   </head>
   <body>
      <c:set var="number" value="42"/>
      <c:choose>
         <c:when test="${ number > 60 }" >
            <c:out value="число ${ number } больше 60"/>
         </c:when>
         <c:when test="${ number > 40 }" >
            <c:out value="число ${ number } больше 40"/>
         </c:when>
         <c:when test="${ number > 10 }" >
            <c:out value="число ${ number } больше 10"/>
         </c:when>
         <c:otherwise>
            <c:out value="число ${ number } меньше 10"/>
         </c:otherwise>
      </c:choose>
   </body>
</html>
```

Если необходимо повторить функциональность if-then-else, следует использовать в теге **<c:choose>** только один **<c:when>** вместе с **<c:otherwise>**.

1.4.3. Итераторы <c:forEach> и <c:forTokens>

Возможности EL предоставляют способ доступа к элементам массивов, списков, множеств, карт отображений, свойств и других итерируемых объек-тов. Однако часто возникает задача вывести всю коллекцию или ее часть, кото-рая может оказаться значительной по размеру. К тому же, в следующем запросе эта коллекция может иметь уже другой размер. Для корректной работы с таки-ми объектами используется тег **<c:forEach>**.

Пусть к экземпляру запроса в методе **doGet()** сервлета **BaseServlet** добав-лен атрибут **lst**, представляющий список:

```
        MessageList messages = new MessageList();
request.setAttribute("lst", messages);
request.getRequestDispatcher("jstl_foreach.jsp").forward(request, response);
```

где экземпляр класса **MessageList** предварительно заполнен значениями:

```
    public class MessageList extends ArrayList<Message> {
  { // логический блок вставлен для упрощения создания коллекции
        // в реальном коде его быть не должно!
    this.add(new Message(72, "Hello"));
    this.add(new Message(31, "Bye"));
    this.add(new Message(11, "Good Bye"));
    this.add(new Message(37, "Wow"));
    this.add(new Message(92, "Hey"));
    this.add(new Message(77, "Hi"));
    this.add(new Message(71, "Ok"));
  }
}
```

Тег **<c:forEach>** имеет атрибуты **items** для размещения коллекции и **var** для доступа к элементу коллекции на каждой итерации. Необязательный атрибут **varStatus** содержит номер итерации, доступ к которому осуществляется мето-дом **getCount()**:

 # 8 # демонстрация работы тегов c:forEach # jstl_fora

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core: forEach</title>
  </head>
  <body>
    <table>
      <c:forEach var="elem" items="${lst}" varStatus="status">
        <tr>
          <td><c:out value="${ elem }" /></td>
          <td><c:out value="${ elem.id }" /></td>
          <td><c:out value="${ elem.text }" /></td>
          <td><c:out value="${ status.count }" /></td>
        </tr>
      </c:forEach>
```

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

В результате в браузер будет выведено (см. рисунок 20).

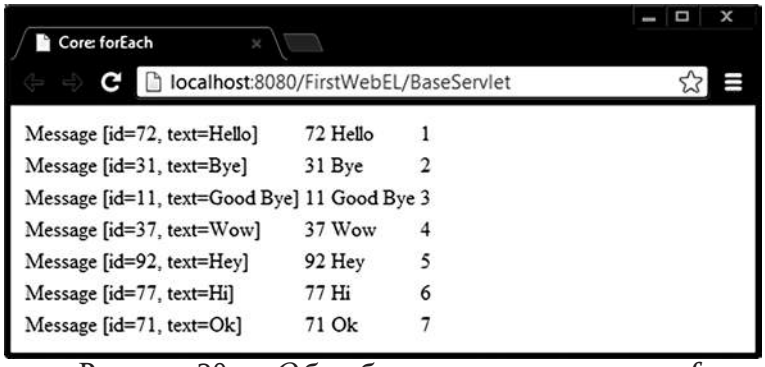


Рисунок 20 - *Обработка списка в теге foreach*

Цикл может начинаться не с первого элемента коллекции и заканчиваться не последним. Атрибуты **begin** (должен принимать значение >=0) и **end** (должен быть >= **begin**) обеспечивают ограниченное действие цикла. Тег **<c:forEach>** в виде

```
<c:forEach var="elem" items="${ lst }" varStatus="status" begin="2" end="4">
```

Изменит вывод предыдущего примера на следующий.

Также можно изменять размер шага движения по коллекции, то есть выводить, например, через один, два или более элементов. Такое использование позволяет организовать атрибут **step**.

```
<c:forEach var="elem" items="${ lst }" varStatus="status" step="2">
```

Вывод исходного примера изменится в очередной раз.

Атрибут **varStatus** в каждом из случаев будет содержать относительную нумерацию в порядке доступа к элементам в цикле вне зависимости от их порядка в самой коллекции.

Если необходимо разбить на части объект типа **String** с использованием разделителей, применяется тег **<c:forTokens>**. Кроме атрибутов **var** и **items**

с аналогичными свойствами здесь применяется атрибут **delims** для определения списка символов-разделителей, отбрасываемых при выделении подстрок. При его отсутствии строка не разбивается.

 # 9 # разбиение строки по шаблону # token.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core: forTokens</title>
  </head>
  <body>
    <c:set var="str" value="45, 76 - 32 : 77 . 91" />
    <c:forTokens var="token" items="${ str }" delims=".,-:)">
      <c:out value="${ token }" />
      <br/>
    </c:forTokens>
  </body>
</html>
```

Результатом выполнения будет вывод в браузер столбца чисел:

45 76 32 77 91

Регулярные выражения для определения разделителей не применяются.

1.4.4. Включение ресурсов

В реальных проектах JSP-страницы часто состоят из набора статических и динамических элементов, находящихся в разных местах контекста и вне его. Заголовок всех страниц приложения (header) и нижняя часть (footer) могут быть одинаковыми. Чтобы избежать плохой практики «copy-paste», следует выделить повторяющиеся части в отдельные страницы или фрагменты страниц и импортировать по необходимости.

Тег **<c:import>** позволяет включать содержимое ресурса, указанного в единственном обязательном атрибуте **url**. Адрес может быть внешним или внутренним, относительным или абсолютным по отношению к данному контексту.

Примеры использования:

- <c:**import** url="\WEB-INF\jspf\footer.jspf" charEncoding="utf-8"/ >
- <c:**import** url="\jspf\header.jspf" charEncoding="utf-8" >
  <param name="title" value="import header info"/>
  <c:**import/**>

- <c:**import** url="ftp://somestore.com/project/sample.war"/>
- <c:**import** url="\js\calendar.js" />ег **<c:import>** получает доступ к источнику, чтение информации из которого происходит непосредственно без буферизации. Контент включается построчно в ис-ходную JSP. По сравнению с action-тегом <**jsp:include**> и директивой <%@**include %**> тег **<c:import>** обеспечивает более совершенное включение динамических ресурсов.

Пусть существует некоторое приложение, состоящее из четырех простых страниц: **index.jsp**, **admin.jsp**, **header.jsp** и **footer.jsp**.

Стартовая страница импортирует общие страницы **header.jsp** и **footer.jsp** и содержит простой переход на страницу **admin.jsp**, которая, в свою очередь также импортирует указанные страницы.

 # 10 # стартовая страница с двумя тегами c:import # i

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
  <head><title>Index page</title></head>
  <body>
    <c:import url="jsp\admin\fragment\header.jsp" />
    <br/>
    <a href="jsp\admin\admin.jsp" >GoTo admin page</a>
    <c:import url="jsp\common\footer.jsp" />
  </body>
</html>
```

Если путь к файлу начинается из корневой директории проекта, то перед первой папкой не следует ставить символ «\» текущего контекста.

Страницы **header.jsp** и **footer.jsp** содержат выражение для извлечения из области видимости запроса значения **admin**. Это значение никто, ни в одной из приведенных страниц не добавлял. Значению в итоге будет присвоен **null**, которое поток вывода игнорирует.

 # 11 # импортируемая страница заголовка # header.jsp

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<html>
  <body>
    <hr/>
 Hello, ${requestScope.admin}
 <hr/>
  </body>
```

```
</html>
```

 # 12 # импортируемая страница с нижней частью #

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<html>
  <body>
      <hr/>
    Copyright, 2001-2020, ${requestScope.admin}
  </body>
</html>
```

Страница **admin.jsp** содержит тег **<c:set>** для добавления атрибута **admin** в область видимости запроса. При импорте страницы получат доступ к этому значению и осуществят его вывод.

 # 13 # страница админа с двумя тегами c:import #

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@taglib uri="jakarta.tags.core" prefix="c" %>
<html>
  <head><title>Admin page</title></head>
  <body>
    <c:set var="admin" value="Blinov" scope="request"/>
    <c:import url="fragment\header.jsp" />
    <form action="${pageContext.request.contextPath}/index.jsp">
      Content admin page
      <br/>
      <input type="submit" value="to Index">
    </form>
    <c:import url="..\common\footer.jsp" />
  </body>
</html>
```

При возврате на **index.jsp** будет создан новый экземпляр **request**, поэтому на этой странице переменная **admin** существовать не будет.

Кодировка включаемой страницы по умолчанию всегда имеет значение ISO-8859-1. Если включаемая страница содержит кириллические символы, то кодировку следует указывать явно при импорте:

```
<c:import url="fragment\header.jsp" charEncoding="utf-8" />
```

1.4.5. Динамические адреса и перенаправление

Запрос можно перенаправить на другую страницу, применив тег **<c:redirect>**. Тег не является часто используемым, так как навигация между страницами при следовании принципам шаблона MVC обычно сильно ограничена.

 # 14 # редирект на другую страницу с передачей пара

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html><body>
      <c:redirect url="jsp/admin/destination.jsp">
        <c:param name="firstname" value="Ostap"/>
        <c:param name="lastname" value="Bender"/>
      </c:redirect>
</body></html>
```

При запуске данной страницы будет сразу же осуществлен переход на стра-ницу **destination.jsp**, и пользователь приложения увидит только результаты ее работы, а именно:

 # 15 # целевая страница с выводом переданных пара

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
  <head><title>Destination page</title></head>
  <body>
    <c:forEach var="ps" items="${param}">
      <c:out value="${ps.key} : ${ps.value}"/><br/>
 </c:forEach>
  </body>
</html>
```

В результате в браузер будут выведены имена переданных параметров и их зна-чения:

lastname: Bender firstname: Ostap

Тег **<c:param>** позволяет объявлять и передавать с запросом параметры, которые доступны на целевой странице и передаются со строкой запроса в виде:

http://localhost:8080/FirstProject/jsp/admin/destination.jsp?firstname=Ostap&lastname=Bender

Тег **<c:url>** позволяет формировать переменную с адресом, значение которого может извлекаться из контекста или формироваться динамически:

 # 16 # динамическое формирование адресов # index

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
  <head>
    <title>Index page</title>
  </head>
  <body>
      <c:url value="jsp/admin/destination.jsp" var="newUrl" />
    <a href='<c:out value="${ newUrl }"/>'>Go</a>
  </body>
</html>
```

1.4.6. JSTL formatting

Библиотека содержит теги форматирования и интернационализации для разработки локализованных приложений. При грамотном использовании страницы вообще не будут содержать текст, а вся информация (надписи на кнопках и ссылках, заголовки, сообщения и пр.) будет извлекаться из файлов свойств.

Теги интернационализации:

<fmt:setLocale/> — устанавливает региональные установки для страницы на основе объекта класса **Locale**;

<fmt:setBundle/>, **<fmt:bundle/>**— устанавливают объект **ResourceBundle**, используемый на странице. В зависимости от установленного значения локали выбирается файл ресурсов (как правило, properties), соответствующий указанному языку, стране или региону;

<fmt:message/> — извлекает локализованное сообщение из ресурса, определенного тегами **<fmt:setBundle/>** или **<fmt:bundle/>**;

<fmt:requestEncoding /> — с атрибутом **value="utf-8"** устанавливает кодировку входящего запроса.

Теги форматирования дат и чисел:

<fmt:timeZone/>, **<fmt:setTimeZone/>** — устанавливает часовой пояс, используемый для форматирования;

<fmt:formatNumber/>, **<fmt:formatDate/>** — форматирует числа/даты с учетом установленной локали (региональных установок) либо указанного шаблона;

<fmt:parseNumber/>, **<fmt:parseDate/>** — переводит строковое представление числа/даты в объекты подклассов **Number** / **Date**.

Ниже приведены три примера на использование тегов из группы **fmt**.

Документ **formatdate.jsp** выводит на экран текущую дату и время с учетом установленного объекта класса **Locale**.

 # 17 # вывод даты и времени # formatdate.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<html>
  <head><title>Data Format</title></head>
  <body>
    <jsp:useBean id="now" class="java.util.Date" />
    <fmt:setLocale value="en-EN"/>
    Вывод даты в формате English<br/>
    Сегодня: <fmt:formatDate value="${now}" /><br/>
    <fmt:setLocale value="be-BY"/>
    Вывод даты в формате Belarus<br/>
    Сегодня: <fmt:formatDate value="${now}" /><br/>
    Стиль времени:
    (short): <fmt:formatDate value="${now}" type="time" timeStyle="short"
/><br/>
    (medium):<fmt:formatDate value="${now}" type="time"
timeStyle="medium"/><br/>
    (long): <fmt:formatDate value="${now}" type="time" timeStyle="long"
/><br/>
  </body>
</html>
```

В следующем примере реализован еще один способ вывода времени и даты

 # 18 # полный вывод даты и времени # timezone.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
```

```
<html>
  <head><title>timezone</title></head>
  <body>
    <jsp:useBean id="now" class="java.util.Date" />
    Вывод даты и времени с помощью тега<br/> fmt:formatDate и установки TimeZone
    <br/>
    <fmt:setLocale value="be-BY"/>
    <fmt:timeZone value="GMT+3:00">
      Locale: BY
        <fmt:formatDate value="${now}" type="both"
      dateStyle="medium"
      timeStyle="long"/><br/>
    </fmt:timeZone>
    <fmt:setLocale value="pl-PL"/>
    <fmt:timeZone value="GMT+1:00">
      Locale: PL :
        <fmt:formatDate value="${now}" type="both" dateStyle="full"
      timeStyle="medium"/><br/>
    </fmt:timeZone>
  </body>
</html>
```

Результат работы страницы смотреть на рисунке 21.

Рисунок 21 - *Вывод даты и времени при задании часовых поясов*

Страница **formatnumber.jsp** выводит числа в соответствии с региональны-ми установками.

 # 19 # форматы числа # formatnumber.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<html>
  <head><title>Format Number</title></head>
  <body>
    <c:set var="currentNumber" value="118000"/>
 <c:out value="Вывод формата числа : ${currentNumber}"/>
    <br/>
    Формат (по умолчанию) : <fmt:formatNumber value="${currentNumber}" />
    <br/>
    Процентный формат : <fmt:formatNumber value="${currentNumber}"
                                          type="percent"/>
    <br/>
    <fmt:setLocale value="be-BY"/>
    Белорусские рубли : <fmt:formatNumber value="${currentNumber}"
                                          type="currency"/>
 <br/>
    <fmt:setLocale value="pl-PL"/>
    Польская валюта :
    <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
    Французская валюта :
    <fmt:setLocale value="fr-FR"/>
    <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
  </body>
</html>
```

Результат работы страницы смотреть на рисунке 21.

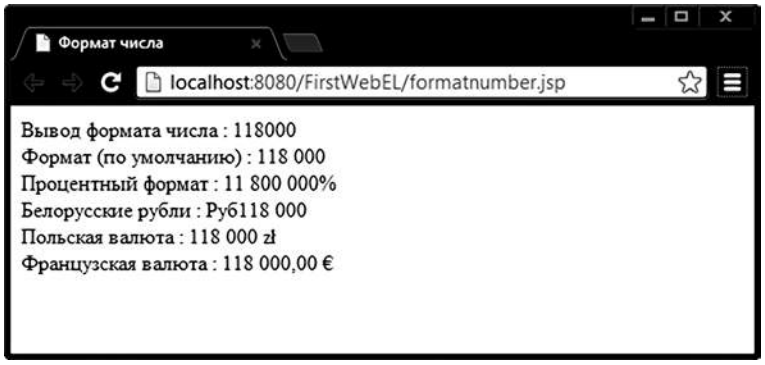


Рисунок 21 - *Обычный, денежный и процентный форматы числа*

Теги **<fmt:setLocale>** и **<fmt:setBundle>** позволяют задать локализацию стра-ницы в целом и исключить из текста страниц конкретные текстовые сообщения, заменяя их ссылками на имена ключей файла ресурсов с расширением **properties**.

 # 20 # локализация страницы с использованием fmt:set

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<fmt:setBundle basename="resource.messages" var="rb" />
  <fmt:setLocale value="en_US" scope="session" /> // value="${locale}"
  <fmt:setBundle basename="prop.pagecontent" var="rb" />
  <html>
    <head>
      <title><fmt:message key="label.title" bundle="${rb}"/></title>
    </head>
    <body>
      <fmt:message key="label.welcome" bundle="${rb}" /> <hr/>
      <fmt:message key="footer.copyright" bundle="${rb}"/>
    </body>
  </html>
```

Результат работы страницы смотреть на рисунке 22.

Где файлы **pagecontent_en_US.properties** и **pagecontent_be_BY.properties** содержат информацию о значении ключей в соответствии с заданным для этих файлов языком и страной:

Рисунок 22 - *Локализованная страница*

```
# Locale en_US
label.title=Example
label.welcome=Welcome!
footer.copyright=Copyright 2001-2013, Blinov
```

а также:

```
# Locale be_BY
label.title=Пример
label.welcome=Добро пожаловать!
footer.copyright=Все права защищены 2001-2013, Блинов
```

Тег **<fmt:setBundle>** задает возможность работы с конкретным файлом ресурсов для всей области видимости страницы.

Тег **<fmt:bundle>** задает возможность работы в своей области видимости с конкретным префиксом, например **label.**, из своей области видимости. То есть информация из ключей, начинающихся с других префиксов, будет недоступна


```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
<fmt:setLocale value="be_BY" scope="session" />
<fmt:bundle basename="prop.pagecontent" prefix = "label." >
   <html>
      <head>
         <title><fmt:message key="title" /></title>
  </head>
  <body>
         <fmt:message key="welcome" />
  </body>
    </html>
</fmt:bundle>
```

Атрибут **prefix** можно опустить, тогда ключи нужно указывать полностью. В результате в браузер будет выведено сообщение:

Добро пожаловать!

Можно попытаться получить доступ к информации с другим ключом:

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
<fmt:setLocale value="ru_RU" scope="session" />
<fmt:bundle basename="prop.pagecontent" prefix = "label." >
  <html>
    <head>
      <title><fmt:message key="title" /></title>
    </head>
    <body>
      <fmt:message key="welcome" /> <br/>
      <fmt:message key="footer.copyright" />
    </body>
  </html>
</fmt:bundle>
```

В результате в браузер уже будет получено:

Добро пожаловать!
??? label. footer.copyright???

При попытке по ключу **footer.copyright** тег **<fmt:message>** пытается найти соответствующее значение с определенным для него префиксом **label** и, не найдя его в файле ресурсов, выдает сообщение **??? label. footer.copyright???, аналогичное сообщение будет выдано , если не найден сам файл ресурсов.**

1.4.7. JSTL sql

Используется для выполнения запросов SQL непосредственно из JSP и обработки результатов запроса в JSP.

<%@taglib uri="jakarta.tags.sql" prefix="sql" %>

— для обычной страницы JSP.

Теги:

<sql:dateParam> — определяет параметры даты для **<sql:query>** либо **<sql:update>**;

<sql:param> — определяет параметры **<sql:query>** либо **<sql:update>**;

<sql:query> — выполняет запрос к БД;

<sql:setDataSource> — устанавливает data source (для пула соединений) для **<sql:query>**, **<sql:update>**, и **<sql:transaction>** тегов;

<sql:transaction> — объединяет внутренние теги **<sql:query>** и **<sql:update>** в одну транзакцию;

<sql:update> — выполняет запрос на вставку/удаление/изменение БД.

В промышленном программировании данная библиотека не используется из-за прямого доступа из JSP в СУБД, что является явным нарушением шаблона MVC.

1.4.8. JSTL xml

Используется для импорта, парсинга и обработки данных из XML-документов в документе JSPX.

Подключение библиотеки для XML формата JSP:

```
<jsp:root version="1.2" xmlns:x= "jakarta.tags.xml">
```

Список тегов:

<x:set> — XML-версия тега **<c:set>;**

<x:out> — XML-версия тега **<c:out>;**

<x:forEach> — XML-версия тега **<c:forEach>**;

<x:if> — XML-версия тега **<c:if>**;

<x:choose> — XML-версия тега **<c:choose>**;

<x:when> — XML-версия тега **<c:when>; <x:otherwise>** — XML-версия тега **<c:otherwise>**; **<x:parse>** — разбор XML-документа;

<x:transform> — трансформация XML-документа с применением XSLT-преобразования;

<x:param> — XML-версия тега **<c:param>**, определяющая параметры для другого тега **<x:transform>**.

XML-документ может быть импортирован из внешнего файла или может быть размещен непосредственно в коде JSP, что практикуется крайне редко.

Тег **<x:parse var="doc">** выполняет разбор документа, представленного в виде тела тега, и помещает построенный объект в переменную **doc**, объявлен-ную в атрибуте **var** тега **parse**.

В XML-документе представлена информация о нескольких студентах, при-чем данные об одном студенте находятся в теге **<student>**. Вывод информации о всех студентах производится тегом

```
<x:forEach select="$doc/students/student" var="stud">
```

где элемент **<student>** со всем его содержимым ассоциируется с переменной **stud** после выбора из объекта **doc** с помощью простого оператора **$doc/students/student**. Далее на каждой итерации цикла из объекта **stud** выбираются элементы **<x:out select="$stud/name"/>** и атрибуты **<x:out select="$stud/@login"/>**, которые выводятся в браузер тегом **<x:out>**.

 # 24 # разбор внедренного документа # xml inside

```
<jsp:root xmlns:jsp=http://java.sun.com/JSP/Page
 xmlns:x="jakarta.tags.xml" version="2.0">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <html>
    <head><title>XML inside</title></head>
    <body>
      <!--using tag x:parse-->
      <x:parse var="doc">
      <students>
        <student login="mit" faculty="mmf">
          <name>Mitar Alex</name>
          <telephone>2456474</telephone>
          <address>
            <country>Belarus</country>
            <city>Minsk</city>
            <street>Kalinovsky 45</street>
          </address>
        </student>
        <student login="pus" faculty="mmf">
          <name>Pashkun Alex</name>
          <telephone>3453789</telephone>
          <address>
            <country>Belarus</country>
            <city>Brest</city>
            <street>Knorina 56</street>
          </address>
        </student>
      </students>
      </x:parse>
      <!--using tags x:forEach and x:out-->
      <x:forEach select="$doc/students/student" var="stud">
        Name: <x:out select="$stud/name"/><br/>
```

```
                Login: <x:out select="$stud/@login" /><br/>
                Faculty: <x:out select="$stud/@faculty" /><br/>
                Country: <x:out select="$stud/address/country" /><br/>
                City: <x:out select="$stud/address/city" /><br/>
                Street: <x:out select="$stud/address/street" /><br/>
                Telephone: <x:out select="$stud/telephone" /><br/>
                <hr/><br/>
              </x:forEach>
            </body>
          </html>
        </jsp:root>
```

Размещение XML-информации непосредственно в документе JSPX снижа-ет гибкость приложения, поэтому разумным решением будет импортировать XML-документ, размещенный вне кода страницы.

 # 25 # извлечение информации из внешнего докумен

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:x="jakarta.tags.xml"
  xmlns:c="jakarta.tags.core" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8" />
    <html>
      <head>
        <title>XML outside</title>
      </head>
      <body>
        <x:parse var="doc">
          <c:import url="../xml/students.xml" />
        </x:parse>
        <!--using tag x:set-->
       Student name:<x:set var="studentName"
select="$doc/students/student[1]/name"/>
          <x:out select="$studentName/." />
      </body>
    </html>
</jsp:root>
```

Результатом будет вывод в браузер строки:

Student name: Mitar Alex

Тег выбор по условию **<x:if>** работает аналогично тегу из библиотеки **core**, только использует свой синтаксис поиска соответствия в экземпляре документа.

 # 26 # поиск информации # xml_if.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:x="jakarta.tags.xml"
  xmlns:c="jakarta.tags.core" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8"/>
    <html>
      <head><title>XML x:if</title></head>
      <body>
        <x:parse var="doc">
          <c:import url="../xml/students.xml"/>
        </x:parse>
        <!--using x:if tag-->
        Find students from Minsk:
        <x:forEach select="$doc/students/student" var="stud">
          <x:if select="$stud/address/city = 'Minsk' ">
                <x:out select="$stud/name" /><br/>
          </x:if>
        </x:forEach>
      </body>
    </html>
</jsp:root>
```

Результатом будет вывод в браузер информации о результатах поиска:

Find students from Minsk: Mitar Alex

Тег **<x:choose>** также аналогичен тегу из базовой библиотеки:

 # 27 # множественный выбор # xml_choose.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:x="jakarta.tags.xml"
  xmlns:c="jakarta.tags.core" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8"/>
    <html>
      <head><title>XML choose</title></head>
      <body>
```

```
<c:import url="../xml/students.xml" var="studXML"/>
<x:parse var="doc" doc="${studXML}"/>
<!--using tags x:choose -->
<x:forEach select="$doc/students/student" var="stud">
  <x:out select="$stud/name" />:
  <x:choose>
    <x:when select="$stud/address/city = 'Minsk'">
      This student from Minsk
    </x:when>
    <x:when select="$stud/address/city = 'Brest'">
      This student from Brest
    </x:when>
  <x:otherwise>
      This student not from Minsk and Brest
    </x:otherwise>
  </x:choose>

<br/>
    </x:forEach>
  </body>
</html>
</jsp:root>
```

Рисунок 23 - *Выбор информации по нескольким условиям*

Результатом будет вывод в браузер (см. рисунок 23).

Преобразование XML-документа в другую форму документа выполняется тегом **<x:transform>**. Для трансформации используется XSL-таблица. В приведенном примере XML-документ преобразуется в HTML-форму с таблицей, в которую выбирается часть информации из XML.

 # 28 # XSL-трансформация # xml_transform.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
```

```
        xmlns:x=jakarta.tags.xml xmlns:c="jakarta.tags.core" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8"/>
    <html>
      <head>
        <title>XML transform</title>
      </head>
      <body>

      <!--using tag x:transform-->

        <c:import url="../xml/students.xml" var="studXML"/>
        <c:import url="../xml/studentsXSL.xsl" var="studXSL"/>
        <x:transform doc="${studXML}" xslt="${studXSL}"/>
      </body>
    </html>
</jsp:root>
```

 # 29 # правило трансформации # studentsXSL.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
    <html>
      <body>

          <table border="0" bgcolor="white">
        <tr>
          <td>Name</td>
          <td>City</td>
        </tr>
        <xsl:for-each select="students/student">
            <tr>
              <td><xsl:value-of select="name" /></td>
                <td><xsl:value-of select="address/city"/></td>
            </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
```

```
    </xsl:template>
</xsl:stylesheet>
```

Результатом будет вывод таблицы в браузер (см. рисунок 24).

Рисунок 24 - *XSL трансформация XML-документа*

1.4.9. JSTL functions

Теги из этой библиотеки выполняют роль, подобную смыслу библиотеки **fmt**, только не для чисел и дат, а для строк. Теги библиотеки используют пре-фикс **fn** и во многом копируют известные методы класса **String** и не составляют особой сложности в применении.

Подключение библиотеки осуществляется директивой **taglib**:

```
<%@ taglib prefix="fn" uri="jakarta.tags.functions" %>
```

Список тегов:

${fn:length(*arg*)} — подсчитывает число элементов в коллекции или длину строки;

${fn:toUpperCase(String str)}** и **${fn:toLowerCase(String str)} — изменяет регистр строки;

${fn:substring(String str, int from, int to)}** — извлекает подстроку; **${fn:substringAfter(String str, String after)} и **${fn:substringBefore(String str, String before)}** — извлекает подстроку до или после указанной во втором аргументе;

${fn:trim(String str)}** — обрезает все пробелы по краям строки; **${fn:replace(String str, String str1, String str2)} — заменяет все вхождения

строки **str1** на строку **str2**;

${fn:split(String str, String delim)} — разбивает строку на подстроки, используя **delim**, как разделитель;

${fn:join(String[] arr, String delim)} — соединяет массив в строку, вставляя между элементами подстроку **delim**;

${fn:escapeXml(String xmlString)} — удаляет из обрабатывемой строки теги;

${fn:indexOf(String str, String searchString)} — возвращает индекс перво-го вхождения строки **searchString**;

<c:if test="${fn:startsWith(String str, String part)}"> — возвращает исти-ну, если строка начинается с подстроки **part**;

<c:if test="${fn:endsWith(String str, String part)}"> — возвращает истину, если строка заканчивается на подстроку **part**;

<c:if test="${fn:contains(String name, String searchString)}"> — возвращает истину, если строка содержит подстроку **searchString**;

<c:if test="${fn:containsIgnoreCase(String name, String searchString))}"> — возвращает истину, если строка содержит подстроку **searchString**, без учета регистра.

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

2.1. Программа лабораторных занятий

1. Запуск JSP/Servlet/JSP. (Hello world!)
2. Подключение Log4J2.
3. Передача числа от клиента на сервер, возвращение этого числа, умноженного на 2.
4. Добавление артефакта предметной области системы. Занесение в базу данных информации, например, телефонной книги (номер и фамилия).
5. Получение в ответ (в случае успешного добавления) списка всех абонентов\пользователей.
6. Использование страниц об ошибках (404, 500 etc).
7. Авторизация (sign in) и выход (sign out) в/из системы. Регистрационная форма MVC (использовать слои). Использовать шифрование пароля.
8. Валидация данных на клиенте и на сервере.
9. Регистрация пользователя (sign up).
10. Регистрация с подтверждением по почте.
11. Модификация информации (например: редактирование профиля, создание и редактирование товара, заказа и т.д.)
12. Удаление информации (например: отмена заказа, записи из телефонной книги и т.д.)
13. Загрузка\выгрузка медиафайлов на сервер, в том числе и в/из БД.
14. Пул соединений с БД.Применение Proxy.
15. Фильтры. Модификация запроса. Проверка прав пользователя при вызове страницы. Установка кодировки запроса\ответа.
16. События. Конфигурирование пула соединений. Отслеживание действий пользователя. Управление информацией в сессии.
17. Локализация приложения. Изменение всех заголовков, имен и сообщений по нажатию кнопки локализации.
18. Cookie. Хранение промежуточной информации между и во время сессии клиента с приложением.
19. Шифрование паролей.
20. AJAX. Асинхронные запросы.
21. Постраничный вывод данных.
22. Post-Redirect-Get. (Защита от F5)
23. Custom tags library.
24. Разработка unit-тестов, в том числе mock.
25. Telegram bot.

2.2. Разработка информационной системы

Целями задания являются:

закрепление студентами теоретических основ проектирования, создания и использования информационных систем;

получение практических навыков создания проектов с помощью IDE;

знакомство с технологиями использования проектирования и разработки информационных систем;

оформление отчёта о проделанной работе.

Поставленные цели достигаются студентами путем проектирования и информационных систем, позволяющей автоматизировать отдельные внутренние и внешние бизнес-процессы некоторой организации (согласно выбранной студентами тематике).

Выполнение задания предполагает несколько последовательных шагов, соответствующих основным этапам жизненного цикла разработки информационных систем:

1) проектирование — на данном этапе определяется предметная область (часть реального мира, которая подлежит изучению) и на ее основе строится информационно-логическая модель (таблицы вариантов использования);

2) создание — на основе информационно-логической модели с помощью любого выбранного студентом средства разработки;

Результаты первого, второго и второго этапов заносятся в письменный отчет к заданию, который должен содержать:

Титульный лист с указанием названия проекта, фамилии, имени и учебной группы автора, а также года выполнения.

Постановку задачи.

Отчёт должен быть подготовлен в MS Word или Google Docs с использованием стилей, созданием автоматически обновляемого оглавления. В конце отчёта привести список список использованных источников (не менее двух печатных изданий и двух электронных изданий),

К заданию, кроме отчёта прикрепить файл с кодом приложения.

При защите проекта обязательным является:

знание и правильное употребление основных терминов, используемых при выполнении задания;

знание потенциальных возможностей;

умение объяснять взаимосвязь всех сущностей и отношений, присутствующих в информационно-логической модели, и объектов, созданных на их основе.

Примерный перечень предметных областей для разработки инфрмационной системы:

Построить веб-систему, поддерживающую заданную функциональность: — на основе сущностей предметной области создать классы, их описывающие; — классы и методы должны иметь названия отражающие их функциональность, и быть грамотно структурированы по пакетам;

— оформление кода должно соответствовать Java Code Convention;

— информацию о предметной области хранить в БД, для доступа использовать API JDBC с использованием пула соединений, стандартного или разработанного самостоятельно. В качестве СУБД рекомендуется MySQL или Derby;

— приложение должно поддерживать работу с кириллицей (быть многоязычной), в том числе и при хранении информации в БД;

— архитектура приложения должна соответствовать шаблону Model-View-Controller;

— при реализации алгоритмов бизнес-логики использовать шаблоны GoF: Factory Method, Command, Builder, Strategy, State, Observer etc.

— используя сервлеты и JSP, реализовать функции, предложенные в постанов-ке конкретной задачи;

— в страницах JSP применять библиотеку JSTL;

— при разработке бизнес-логики использовать сессии и фильтры;

— использовать Log4j;

— код должен содержать комментарии.

1. Система **Факультатив**. Существует перечень **Курсов,** за каждым из которых закреплен один **Преподаватель**. **Студент** записывается на один или несколько **Курсов**. По окончании обучения **Преподаватель** выставляет **Студенту** и добавляет отзыв.
2. Система **Платежи**. **Клиент** имеет одну или несколько **Кредитных Карт,** каждая из которых соответствует некоторому **Счету** в системе платежей. **Клиент** может при помощи **Счета** сделать **Платеж**, заблокировать **Счет** и пополнить **Счет**. **Администратор** снимает блокировку.
3. Система **Приемная комиссия**. **Абитуриент** регистрируется на один из **Факультетов** с фиксированным планом набора, вводит баллы по соответствующим **Предметам и аттестату**. Результаты **Администратором** регистрируются в **Ведомости**. Система подсчитывает сумму баллов и определяет **Абитуриентов**, зачисленных в учебное заведение.
4. Система **Библиотека**. **Читатель** имеет возможность осуществлять поиск и заказ **Книг** в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. **Книга** может присутствовать в **Библиотеке** в одном или нескольких экземплярах.

5. Система **Больница**. **Врач** определяет диагноз, делает назначение **Пациенту** (процедуры, лекарства, операции). Назначение может выполнить **Медсестра (**процедуры, лекарств**а)** или **Врач** (любое назначение). **Пациент** может быть выписан из **Больницы**, при этом фиксируется окончательный диагноз.
6. Система **Турагентство**. **Заказчик** выбирает и оплачивает **Тур (отдых, экс-курсия, шоппинг)**. **Турагент** определяет тур как «горящий», размеры ски-док постоянным клиентам.
7. Система **Телефонная станция**. **Администратор** осуществляет подключе-ние **Абонентов**. **Абонент** может выбрать одну или несколько из предостав-ляемых **Услуг**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**. **Администратор** может просмотреть список неоплаченных **Счетов** и за-блокировать **Абонента**.
8. Система **Автобаза**. **Диспетчер** распределяет **Заявки** на **Рейсы** между **Водителями,** за каждым из которых закреплен свой **Автомобиль**. На **Рейс** может быть назначен **Автомобиль**, находящийся в исправном состоянии и характеристики которого соответствуют **Заявке. Водитель** делает отмет-ку о выполнении **Рейса** и состоянии **Автомобиля**.
9. Система **Интернет-магазин**. **Администратор** осуществляет ведение ката-лога **Товаров**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** может занести неплательщиков в «черный список».
10. Система **Авиакомпания**. **Авиакомпания** имеет список рейсов. **Диспетчер** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. **Администратор** управляет списком рейсов.
11. Система **LowCost-Авиакомпания**. **Клиент** заказывает и оплачивает **Билет** на **Рейс** с учетом наличия/отсутствия багажа и права первоочередной реги-страции и посадки (Цена **Билета** может быть ниже стоимости провоза ба-гажа). С приближением даты **Рейса** или наполнением самолета цена на **Билет** может повышаться.
12. Система **Периодические издания**. **Администратор** осуществляет ведение каталога периодических **Изданий**. **Читатель** может оформить **Подписку**, предварительно выбрав периодические **Издания** из списка. Система под-считывает стоимость и регистрирует **Платеж**.
13. Система **Заказ гостиницы**. **Клиент** заполняет **Заявку**, указывая количест-во мест в номере, класс апартаментов и время пребывания. **Администратор** просматривает поступившую **Заявку**, выделяет наиболее подходящий из доступных **Номеров**, после чего система выставляет **Счет Клиенту**.
14. Система **Жилищно-коммунальные услуги**. **Квартиросъемщик** отправ-ляет **Заявку**, в которой указывает род работ, масштаб и желаемое время выполнения. **Диспетчер** формирует соответствующую **Бригаду** и реги-стрирует её в **Плане работ**.

15. Система **Прокат автомобилей**. **Клиент** выбирает **Автомобиль** из списка доступных. Заполняет форму **Заказа**, указывая паспортные данные, срок аренды. **Клиент** оплачивает **Заказ**. **Администратор** регистрирует возврат автомобиля. В случае повреждения **Автомобиля Администратор** вносит информацию и выставляет счет за ремонт. **Администратор** может откло-нить **Заявку**, указав причины отказа.
16. Система **Скачки. Клиент** делает **Ставки** разных видов на **Забеги**. **Букмекер** устанавливает уровень выигрыша. **Администратор** фиксирует результаты **Забегов**.
17. Система **Тестирование**. **Тьютор** создает **Тест** из нескольких **Вопросов** закрытого типа (выбор одного или более вариантов из N предложенных) по определенному **Предмету**. **Студент** просматривает список доступных **Тестов,** отвечает на **Вопросы**.
18. Система **Ресторан**. **Клиент** осуществляет заказ из **Меню**. **Администратор** подтверждает **Заказ** и отправляет его на кухню для исполнения. **Администратор** выставляет **Счет**. **Клиент** производит его оплату.
19. Система **Кофе-машина**. **Пользователь** обладает **Счетом**. **Кофе-машина** содержит набор **Напитков**, с заданным числом порций и дополнительных **Ингредиентов**. **Пользователь** может купить один или несколько **Напитков**. **Администратор Кофе-машины** осуществляет ее наполнение.
20. Система **Парк**. **Владелец** парка дает указания **Леснику** о высадке (лечении, художественной обработке, уничтожении) **Растений**. **Лесник** отчитывается о выполнении. **Владелец** просматривает результаты и подтверждает исполнение.
21. Система **Команда разработчиков**. **Заказчик** представляет **Техническое Задание (ТЗ)**, в котором перечислен перечень **Работ** с указанием квалификации и количества требуемых специалистов. **Менеджер** рассматривает **ТЗ** и оформляет **Проект**, назначая на него незанятых **Разработчиков** требуе-мой квалификации, после чего рассчитывается стоимость **Проекта** и **Заказчику** выставляется **Счет**. **Разработчик** имеет возможность отме-тить количество часов, затраченных на работу над проектом.
22. Система **Железнодорожная касса***. **Пассажир** делает **Заявку** на билет до необходимой ему станции назначения, время и дату поездки. Система осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** управляет списком зарегистрированных пассажиров.
23. Система **Городской транспорт***. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом или расписанием.

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1. Перечень рекомендуемых средств диагностики

Диагностика результатов учебной деятельности по дисциплине «Разработка Веб-приложений» проводится, как правило, во время аудиторных занятий. Для диагностики используются:

– устный опрос;

– отчет по лабораторным и домашним заданиям.

Контроль УСР проводится преподавателем с использованием ИКТ в форме опроса и проверки результатов выполнения работы.

Формой промежуточной аттестации по дисциплине «Разработка Веб-приложений» учебным планом предусмотрен **зачет.**

3.2. Примерный перечень заданий для управляемой самостоятельной работы студентов

Установка и конфигурирование IDE. (2ч)

Задание 1. Загрузка и установка IDE

Форма контроля – опрос, отчет по лабораторным / домашним заданиям.

3.3. Описание инновационных подходов и методов к преподаванию учебной дисциплины

При организации образовательного процесса используются

- ***методы и приемы развития критического мышления,*** которые представляют собой систему, формирующую навыки работы с информацией в процессе чтения и письма; понимании информации как отправного, а не конечного пункта критического мышления;
- ***практико-ориентированный подход,*** который предполагает освоение содержание образования через решения практических задач.

3.4. Методические рекомендации по организации самостоятельной работы обучающихся

При изучении учебной дисциплины рекомендуется использовать следующие формы самостоятельной работы:

- изучение литературы и материалов электронных источников по проблемам дисциплины;
- выполнение домашнего задания;

- подготовка к лабораторным занятиям;
- курсовые, дипломные и научно-исследовательские работы, связанные с тематикой дисциплины;
- подготовка к участию в конференциях с докладами по проблемам дисциплины.

Для организации самостоятельной работы студентов по учебной дисциплине рекомендуется использовать современные информационные ресурсы, размещенные на образовательном портале смешанного и дистанционного обучения БГУ, содержащие учебные материалы (курс лекций, задания к лабораторным и домашним работам, перечень вопросов к зачёту и т. д.)

3.5. Примерный перечень вопросов к зачету

1. Что такое Java-сервлеты и как они связаны с веб-приложениями?
2. Как создать Java-сервлет и какие методы жизненного цикла они имеют?
3. Что такое контейнер сервлетов и как он обрабатывает запросы и ответы?
4. Как получить параметры запроса в Java-сервлете?
5. Как передать данные между сервлетами?
6. Что такое JSP (JavaServer Pages) и как они используются в веб-приложениях?
7. Как объявить и использовать переменные в JSP?
8. Что такое скриптлеты в JSP и как они используются?
9. Как использовать выражения JSP для вывода данных на веб-страницу?
10. Какие директивы JSP вы знаете и для чего они нужны?
11. Что такое JSP-теги и как они используются?
12. Какие стандартные теги JSTL (JSP Standard Tag Library) вы знаете и для чего они предназначены?
13. Как использовать циклы и условные операторы в JSP с помощью JSTL?
14. Как использовать JSTL для работы с коллекциями данных?
15. Как обрабатывать исключения в сервлетах и JSP?
16. Как реализовать сессии в Java-сервлетах?
17. Какие методы HTTP запросов вы знаете и как их использовать в сервлетах?
18. Как отправить перенаправление (redirect) в сервлете?
19. Как работать с файлами в сервлетах?
20. Как обрабатывать формы HTML в сервлетах?
21. Как работать с куками (cookies) в сервлетах?
22. Как обеспечить безопасность веб-приложения с помощью сервлетов и JSP?
23. Как использовать базы данных в сервлетах и JSP?
24. Какие средства тестирования и отладки вы знаете для сервлетов и JSP?

25. Как работать с сеансами (sessions) в JSP?
26. Как ограничить доступ к определенным страницам или функциональности в веб-приложении?
27. Как обрабатывать загрузку файлов в сервлетах?
28. Как использовать фильтры (filters) в сервлетах?
29. Как обрабатывать AJAX-запросы в сервлетах и JSP?
30. Как развернуть веб-приложение, содержащее сервлеты и JSP, на сервере приложений?
31. Что делает RequestDispatcher.include()?
32. В какой последовательности выполняются сервлет фильтры?
33. Как обрабатывается тег с телом?
34. Что нужно написать в строке браузера, чтобы обратиться к хосту, на котором установлен tomcat, развёрнуто приложение, в котором есть несколько сервлетов? Как обратиться к конкретному сервлету? Что такое www? Где нужно указывать порт?
35. Можно ли в web.xml определить сервлет без указания url паттерна и как к нему обратиться?
36. Что такое HTTP? Отличия HTTP 1.0 и HTTP 2.
37. Сохраняет ли http протокол своё состояние.
38. Как сервер понимает, что для пользователя создана сессия и не нужно её создавать?
39. Отличие авторизации от аутентификации.

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1. Список рекомендуемой литературы

Основная литература

1. Эккель, Брюс. Философия Java = Thinking in Java / Брюс Эккель ; [пер. с англ. Е. Матвеев]. - 4-е полное изд. - Санкт-Петербург [и др.] : Питер, 2023. - 1168 с.
2. Блинов, И. Н. Java from EPAM : учебно-методическое пособие [для студентов высших учебных заведений, обучающихся по специальностям 1-31 03 08 "Математика и информационные технологии (по направлениям)", 1-31 03 01 "Математика (по направлениям)"] / И. Н. Блинов, В. С. Романчик. - Минск : Четыре четверти : EPAM Training Center, 2020. - 559 с.
3. Белугина, С. В. Разработка программных модулей программного обеспечения для компьютерных систем. Прикладное программирование : учебное пособие / С. В. Белугина. — Санкт-Петербург : Лань, 2020. — 312 с. — Текст : электронный // Лань : электронно-библиотечная система. — [Электронный ресурс]. – Режим доступа: https://e.lanbook.com/book/133920.
4. Гонсалвес, Э. Изучаем Java EE 7. - Санкт-Петербург [и др.] : Питер, 2018. - 640 с.

Дополнительная литература

1. Блинов И.Н., Романчик В.С. Методы программирования Java. – Четыре четверти, 2013. – 896 с.
2. Хорстманн К. С., Корнелл Г. Библиотека профессионала. Java 2. Том 1. Основы. – М.: “Вильямс”, 2019. – 866 с.
3. Блинов И.Н., Романчик В.С. Java. Промышленное программирование. – Универсалпресс, 2007. – 704 с.

4.2. Учебно-методическая карта учебной дисциплины

Дневная форма получения образования

Учебно-методическая карта

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов УСР	Формы контроля знаний
		Лекции	Практические	Лабораторные	Иное		
	2	3	4	5	6	7	8

	Обзор технологий JEE	2		2			опрос
	Основы клиент-серверных систем	4		4			Опрос, отчет по лабораторным / домашним работам
	Паттерны проектирования	4		4			Опрос
	Сервлеты	6		4			Опрос, отчет по лабораторным / домашним работам
	ORM-технологии доступа к базами данных (JPA, JDBC)	4		4			Опрос
	JSP и JSF	6		4			Опрос, отчет по лабораторным / домашним работам
	AJAX	4		4			Опрос
	Технология Spring MVC	4		4			Опрос, отчет по лабораторным / домашним работам
	ВСЕГО ЧАСОВ	**34**	**-**	**30**			**Зачет**

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Заочная форма получения образования

Учебно-методическая карта

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов				Количество часов УСР	Формы контроля знаний
		Лекции	Практические (семинарские)	Лабораторные занятия	Иное		
	2		4	5	6	7	8

-2	Обзор технологий JEE. Основы клиент-серверных систем			**0,5**			Опрос
	Паттерны проектирования			0,5			Опрос
	Сервлеты	, 5		1			Опрос
	ORM-технологии доступа к базами данных (JPA, JDBC)	, 5		1			Опрос
	JSP и JSF			1			Опрос
	AJAX			1			Опрос
	Технология Spring MVC			1			Опрос, отчет по лабораторным / домашним работам
	ВСЕГО ЧАСОВ		**-**	**6**			**Зачет**