Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

РАСПРЕДЕЛЕННЫЕ СЕРВИС-ОРИЕНТИРОВАННЫЕ СИСТЕМЫ

Учебное пособие

Томск
2020

УДК 004.75
ББК 30.2-5-05
Р 344

Рецензенты:

Бойченко И.В., программист АО «ИнфоТеКС», кандидат техн. наук

Ефремов В.А., ведущий инженер ООО «КС Групп», кандидат техн. наук

Резник, Виталий Григорьевич

Р 344 Распределенные сервис-ориентированные системы. Учебное пособие / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2020. – 305 с.

Учебное пособие предназначено для обучения дисциплине «Распределенные сервис-ориентированные системы» для студентов направления подготовки магистратуры: 09.04.01 «Информатика и вычислительная техника», направленность (профиль) программы - «Программное обеспечение вычислительных машин, систем и компьютерных сетей».

Одобрено на заседании каф. АСУ протокол №_5_ от _22.04.2021_

УДК 004.75
ББК 30.2-5-05

© Резник В. Г., 2020
© Томск. гос. ун-т систем упр. и радиоэлектроники, 2020

Введение

Данное учебное пособие содержит учебный материал предназначенный для студентов магистратуры по дисциплине «*Распределенные сервис-ориентированные системы*» (РСОС) для направления подготовки 09.04.01 «Информатика и вычислительная техника».

Предметная область изучаемой дисциплины является составной частью более обширного научно-технического направления, часто обозначаемого как «*Распределенные вычислительные системы*» и изучаемого в рамках направления подготовки 09.03.01 на уровне бакалавриата. Соответственно, принятый в данном пособии стиль изложения предполагает, что студент уже знаком с определениями и аргументацией учебного пособия [1] и использует его в качаестве базовой основы изучаемой дисциплины.

Целью изучаемой дисциплины РСОС является изучение общих архитектурных принципов построения сервис-ориентированных систем, территориально распределенных по множеству вычислительных машин (ВМ) и объединенных как средствами компьютерных сетей, так и стандартизированными средствами программного обеспечения.

Основной задачей изучения дисциплины является формирование у обучающихся теоретических представлений о современных подходах, которые направлены на проектирование элементов РСОС, а также на практическое освоение методов и инструментальных средств, способствующих их успешной реализации.

В процессе обучения студент использует литературные источники, рекомендованные программой обучения по данному курсу, а по результатам обучения он должен:

а) знать общие принципы построения и терминологию описания распределенных сервис-ориентированных систем; теорию и практику предметной области; этапы и технологию проектирования РСОС;

б) уметь самостоятельно разрабатывать программы, реализующие элементы распределенных систем; проводить технологическое описание распределенной предметной области; использовать инструментальные средства реализации элементов РСОС;

в) владеть инструментальными средствами языка Java специализированными на создание распределенных сервисных систем; инструментальными средствами создания веб-сервисов.

Общее содержание дисциплины по главам отражает следующую познавательную тематику:

а) **Тема 1**. Предметная область и терминология РСОС.

б) **Тема 2**. Использование компоненты JSF контейнера Web.

в) **Тема 3**. Современные способы доступа к данным.

г) **Тема 4.** Обработка документов XML и JSON.

д) **Тема 5**. Web-службы SOAP.

е) **Тема 6**. Web-службы в стиле REST.

Глава 1 дает краткое описание изучаемой предметной области. Ее содержание основано на обобщении материала, изложенного в главах 1 и 5 учебного пособия [1], и конкретизировано для целевой предметной области. В ней также раскрывается назначение и тематика последующих глав.

Глава 2, озаглавленная как «*Использование компоненты JSF контейнера Web*», посвящена демонстрации контейнерной и компонентной технологии программной платформы Java EE на примере серверной парадигмы представления визуальной информации.

Глава 3 раскрывает программную платформу Java EE, демонстрируя новые подходы работы с СУБД, отличные от классической парадигмы SQL-запросов. Такими подходами являются EJB-компоненты серверов приложений и технология JPA, обеспечивающая объектно-реляционное отображение объектов-сущностей на структуру таблиц баз данных.

Глава 4 описывает инструментарий платформы Java EE для работы с информацией, представленной форматами документов XML и JSON, что обеспечивает обмен сообщениями в сервис-ориентированных системах.

Глава 5 непосредственно рассматривает классическое представление сервис-ориентированных систем, основанных на использовании протокола SOAP и языка описания Web-сервисов — WSDL.

Глава 6 завершает учебный курс дисциплины, демонстрируя реализацию Web-служб в стиле REST.

В заключении к данному учебному пособию подводятся итоги по всему объему теоретического и практического материала дисциплины «*Распределенные сервис-ориентированные системы*». Эта часть определяет границы усвоенного студентом учебного материала и намечает пути дальнейшего изучения рассмотренной тематики.

1 Тема 1. Предметная область и терминология РСОС

Объектом внимания изучаемой дисциплины «*Распределенные сервис-ориентированные системы*» является особый класс искусственно созданных на основе вычислительной техники систем, элементы которых представляются в виде двух основных групп: *поставщиков сервисов* и *потребителей сервисов*. В последующем изложении текста мы будем использовать акроним **РСОС**, понимая под ним как объект или предмет изучения, так и частные теоретические толкования уже принятые в литературных источниках.

Предметом изучения дисциплины РСОС являются технологии проектирования и реализации подобных систем, получившие достаточную известность к настоящему времени и подкрепленные общепринятыми стандартами международных организаций.

Целью изучения дисциплины РСОС является получение более глубоких знаний и практических навыков о современных технологиях создания сложных компьютерных систем, охватываемых более широкой предметной областью дисциплины «*Распределенные вычислительные системы*».

В качестве результата этой цели предполагается подготовка обучающегося до уровня, обеспечивающего самостоятельное создание элеменов РСОС и возможность его успешной работы в соотвествующих коллективных проектах. Тем не менее, в силу достаточно большого объема и сложности изучаемой предметной области, знания о ней ограничены только технологиями реализованными на основе языка Java. Подобное ограничение имеет целью обеспечить целостность практических навыков обучающихся уже проверенными подходами, доказавшими свою значимость, а также сохранить преемственность учебно-методического подхода, сформированного в учебном пособии [1]. С этой же целью мы будем опираться на сформулированные ранее термины и определения, раскрывая и модифицируя их по мере необходимости.

Сосредоточенные системы группируют «классическое направление» развития цифровых технологий, основаных на вычислительной мощности программно-аппаратных средств и локализованных во вполне обозримых границах. Для таких систем подключение и взаимодействие с другими подобными системами является допустимым явлением, но не главной характеристикой.

Детальное толкование таких систем интерпретируется в трех основных аспектах:

1. **ЭВМ** — согласованный в функциональном назначении аппаратный конструктив, дополненный базовым программным обеспечением (ПО) операционных систем (ОС) до конкретной виртуальной машины; в последующем, такая машина рассматривается как элемент более сложных систем;

2. **Вычислительные системы** — все прикладные аспекты программного обеспечения, реализованные поверх виртуальной машины ОС; в последующем, такие системы также рассматриваются как элементы более сложных систем;
3. **Вычислительные комплексы** — системы аппаратных средств, ориентированные на повышение вычислительной мощности и надежности всего конструктива, а также — дополненные соотвествующей виртуальной машиной ОС; в последующем, такой комплекс также может рассматривается как отдельная ЭВМ или составляющая часть вычислительной системы.

Распределенные системы группируют «*современное направление*» развития цифровых технологий, основной характеристикой которых является интеграция *сосредоточенных систем* (ЭВМ, ВС) на основе сетевых техно-логий. Важной особенностью таких систем является их *свойство несводи-мости* к моделям сосредоточенных систем. Обычно, это определяется самой природой интегрируемых приложений, например, программное обеспечение клиента банка должно быть отделено от ПО самого банка в силу норма-тивных требований к проведению финансовых операций.

Распределенные вычислительные сети (***РВ-сети***) — распределенные системы, дополнительно интерпретируемые схемой рисунка 1.1.

Рисунок 1.1 — Обобщенная модель РВ-сетей

С целью более полного раскрытия терминологии и содержания предмет-

ной области дисциплины, рассмотрим ее место в более широкой области распределенных систем, которую уточним в двух аспектах. Сначала рассмотрим этапы развития РВ-сетей до уровня объектных распределенных систем. Затем опишем становление систем с сервис-ориентированной архитектурой. Это позволит более точно обосновать структуру и последовательность изложения учебного материала данного пособия.

1.1 Этапы развития распределенных систем

Любые получаемые студентом знания отражают прошлое, сформулированное в настоящем и проецируемое на будущее. Такому же отражению подвергаются и технологии изучаемой предметной области. Например, в апреле 1964 года корпорация IBM анонсировала семейство компьютеров класса майнфреймов, обозначив это семейство как IBM System/360 и объявив его как «*компьютеры третьего поколения*». Это событие стимулировало формирование знаний об эволюции средств автоматизации вычислений, что нашло отражение в ряде международных стандартов и ГОСТ как последовательность поколений вычислительных машин (ВМ): начиная с нулевого поколения и заканчивая — шестым. Учебник [2] отражает это событие в виде «…

Вычислительная машина (ВМ) — совокупность технических средств, создающая возможность проведения обработки информации (данных) и получение результата в необходимой форме. Под техническими средствами понимают все оборудование, предназначенное для автоматизированной обработки данных. Как правило, в состав ВМ входит и системное программное обеспечение. …

Поколение вычислительных машин — это классификационная группа ВМ, объединяющая ВМ по используемой технологии реализации ее устройств, а также по уровню развития функциональных устройств и программного обеспечения и характеризующая определенный период в развитии промышленности средств вычислительной техники. ...»

В настоящее время знание характеристик каждого поколения ВМ является обязательным для специалистов научного направления «Информатика и вычислительная техника», тем не менее, как отмечено в том же учебнике [2]: «... Пятое и шестое поколения в эволюции ВТ — это отражение нового качества, возникающего в результате последовательного накопления частных достижений, главным образом в архитектуре вычислительных систем и, в меньшей мере, в сфере технологий ...».

Последнее замечание показывает нам, что шестое поколение ВМ, обозначенное как «*1990 — и далее*», исчерпало себя как текущая парадигма прогресса и полностью уступила место новым идеям, основанным на развитии сетевых технологий и систем. В концептуальном плане это привело к выделению новой и самостоятельной группы систем «*Распределенные системы*», что стало про-

тивоставляться «старой» группе, именуемой как «*Сосредоточенные системы*». Этот аспект разделения и рассмотрим в следующих пунктах.

1.1.1 Классификация систем обработки данных

В период пятого поколения ВМ (1984 — 1990 годы) интенсивно стали распространяться персональные компьютеры на базе микропроцессорной техники. Кроме широкого распространения ОС UNIX появляются: Mac OS (Macintosh Operating System, 1984 год), MS Windows (1985 год), Minix (UNIX-подобная микроядерная ОС, 1987 год). Появляется возможность обучения системным исследованиям с использованием языков программирования, появившихся в период четвертого поколения эволюции ВТ, например, таких как ***C***. В образовательной среде вузов начинает закрепляться понятие «*Системы обработки данных*» или СОД, показанное на рисунке 1.2.

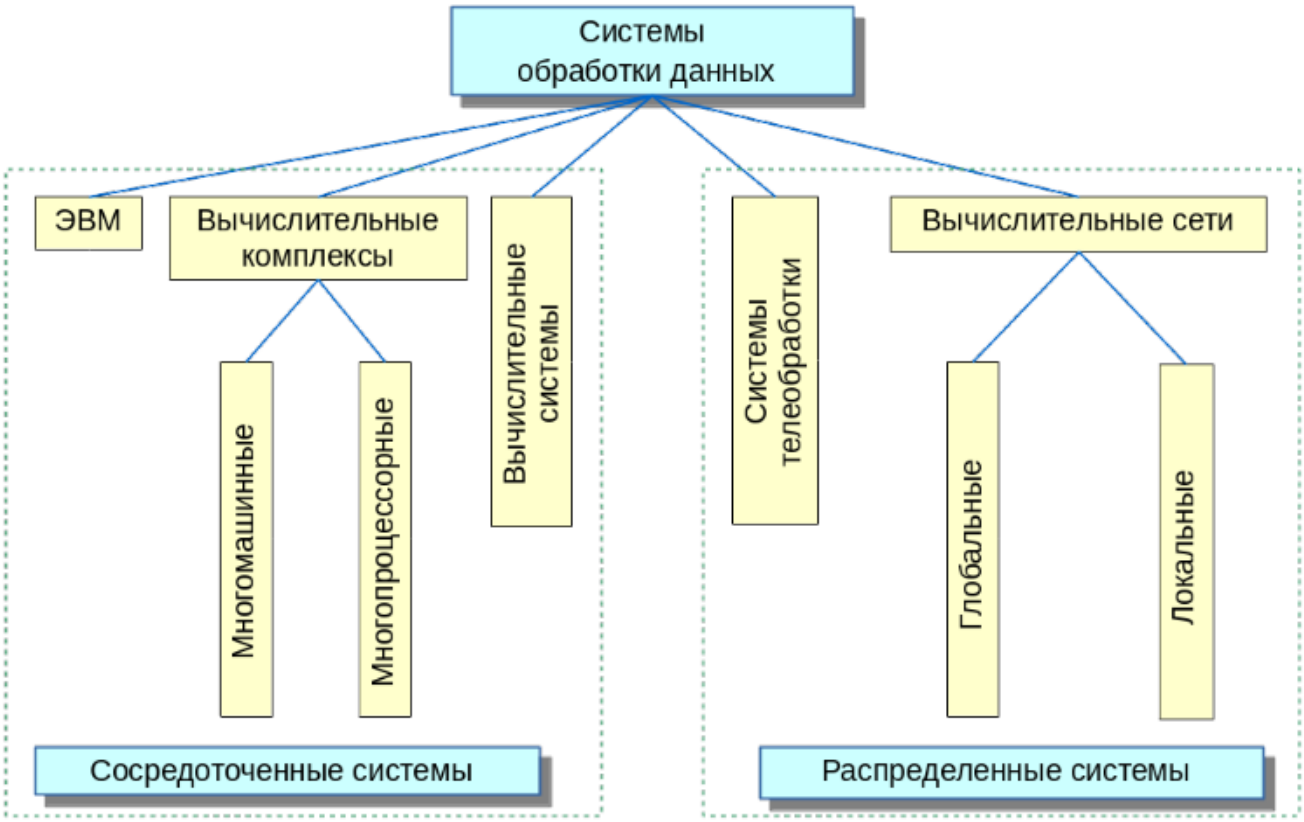

Рисунок 1.2 — Структура классификации СОД (Ларионов А.М., [3])

Здесь хорошо видно разделение на сосредоточенные и распределенные системы, причем предполагается, что все указанные элементы классификации дополняются системным программным обеспечением (ОС), а вычислительные сети характеризуются следующим образом [3, стр. 8-9]: «... **Вычислительные сети**. С ростом масштабов применения электронной вычислительной техники в

научных исследованиях, проектно-конструкторских работах, управлении производством и транспортом и прочих областях стала очевидна необходимость объединения СОД, обслуживающих отдельные предприятия и коллективы. Объединение разрозненных СОД обеспечивает доступ к данным и процедурам их обработки для всех пользователей, связанных общей сферой деятельности ...».

К существенным недостаткам классификации Ларионова следует отнести излишний акцент на технической части сетевых технологий, доступных в период пятого поколения ВТ. Это искажает современное представление о технологиях, применяемых в этой предметной области.

1.1.2 Распределенные вычислительные сети

Достаточно подробная критика классификации Ларионова приведена в учебном пособии автора [1]. Она основана на том, что выделение систем телеобработки, локальных и глобальных сетей отражает лишь предметную область сетевых технологий, а современные распределенные системы характеризуются:

а) разделенными вычислительными ресурсами;

б) поставщиками ресурсов;

в) потребителями ресурсов;

г) средами взаимодействия между поставщиками и потребителями вычислительных ресурсов.

В условиях выделенных характеристик был обоснован и использован термин «*Распределенные вычислительные сети*» (РВ-сети), который конкретизирует предметную область распределенных систем, предполагая интерпретацию:

1. **Вычислительные системы** — элемент разделенных вычислительных ресурсов, имеющий конкретный собственный сетевой адрес;
2. **Поставщики ресурсов** и **Потребители ресурсов** — являются конкретными характеристиками прикладной части распределенных систем;
3. **Среды взаимодействия** — характеристики служебных частей распределенных систем.

В классическом варианте РВ-сети начинали развиваться на основе простейшей модели «*Клиент-сервер*», основанной на методе «*Удаленный вызов процедур*» и схематично показанной на рисунке 1.3.

Удаленный вызов процедур (Remote Procedure Call, RPC) — технология, позволяющая программам одной ВМ вызывать функции или процедуры на другой ВМ. Характерной чертой RPC является работа в разных адресных пространствах, как правило, на разных ВМ. Для согласования адресных пространств необходимо создавать программные заглушки как на стороне клиента,

так и на стороне сервера:

1) **Client stub** — программная заглушка на стороне клинета;
2) **Server stub** — программная заглушка на стороне сервера.

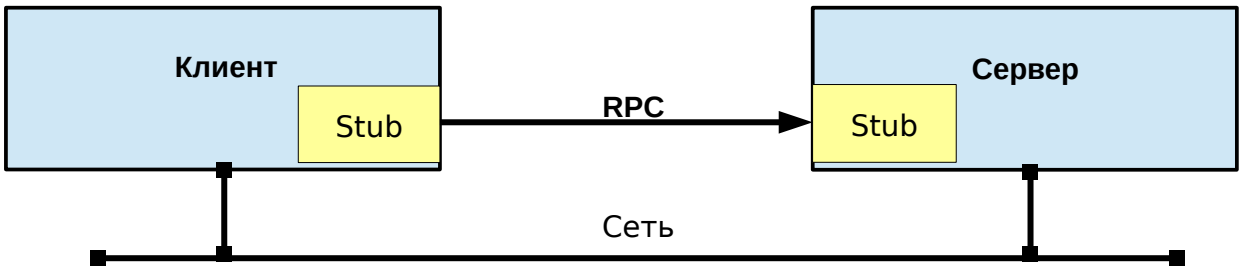

Рисунок 1.3 — Классическая модель взаимодействия «Клиент-сервер»

В начале 90-х годов ряд компаний стала развивать идею распределенных вычислительных сред (***Distributed Computing Environment***, ***DCE***), которые на основе методов RPC должны были включать в себя:

а) удаленный вызов процедур DCE/RPC;

б) службу каталогов;

в) службу связанную со временем;

г) службу проверки подлинности;

д) файловую систему DCE/DFS.

Подобную идею развивал и Эндрю Таннебаум, создавая ***Distributed Operating Systems*** (***Распределенные Операционные Системы***), о чем можно прочитать в его работе [4].

Существенным недостатком классической модели «*Клиент-сервер*», основанной на методе RPC, является необходимость учитывать все детали использования сетевых протоколов транспортного уровня. Это существенно ограничивает масштаб реализаций прикладных систем, делая их сложными и ненадежными.

1.1.3 Объектные распределенные системы

В 1984 году была подготовлена первая версия стандарта Open Systems Interconnection (OSI) — Basic Reference Model: The Basic Model, которая зафиксировала открытую семиуровневую архитектуру протоколов для взаимосвязи приложений в сети.

В 1989 году была организована *Object Management Group* (*OMG*) – рабочая группа, включающая компании Hewlett Packard, Sun Microsystems и Canon, которые стали продвигать объектно-ориентированные технологии и стандарты.

В начале 90-х годов популярными становятся языки объектно-ориентированного программирования (ООП): Ada (1980 год), C++ (1983 год), Object Pascal (1986 год) и другие. Этому способствовали все возрастающие возможности ВМ и потребность в разработке крупных и надежных приложений.

В 1991 году OMG выпустила первую версию стандарта на технологию проектирования и реализации объектных распределенных систем **Common Object Request Broker Architecture** (**CORBA**), в которой сетевая среда взаимодействия приложений обеспечивалась программным обеспечением брокеров. Таким образом стал формироваться промежуточный слой программного обспечения, который в архитекурах DCE обозначался как ***Midlleware***.

В мае 1995 года компания Sun Microsystems выпустила первую версию языка ООП Java. Основываясь на виртуальной машине времени выполнения **JVM** (**Java Virtual Machine**), этот язык обеспечивал почти абсолютную переносимость программного обеспечения между различными платформами ВМ, что является важным свойством для создания распределенных гетерогенных систем. Уже в феврале 1997 года Java версии 1.1 полностью обеспечивала технологию CORBA под собственным брендом **RMI** (**Remote Method Invocation**).

Объектный подход вводит новое понятие «***Распределенный объект***» (***Distributed object***) — удаленный объект, размещенный на ВМ сервера, интерфейсы которого расположены на машинах клиентов.

В целом, стандарты CORBA и RMI значительно продвинули технологии создания РВ-сетей. Детали теории и реализации таких систем достаточно подробно описаны в учебном пособии [1, глава 3]. Здесь отметим лишь наличие общности объектного подхода и классического метода RPC, которая требует реализации на ВМ клиента и ВМ сервера специального ПО обозначаемого как ***stub*** (заместитель, скелетон и другие элементы технологии ***proxy-***). Это демонстрируется рисунком 1.4.

Существенным недостатком объектного подхода является потребность наличия на обоих взаимодействующих сторонах (на клиенте и сервере) описания классов используемых объектов. Такой факт делает такие системы *сильносвязанными*.

С целью обеспечения надежности работы объектного подхода реализации РВ-сетей, взаимодействие ПО клиента и сервера осуществляется через промежуточное ПО называемое ***брокером***, что показано на рисунке 1.5.

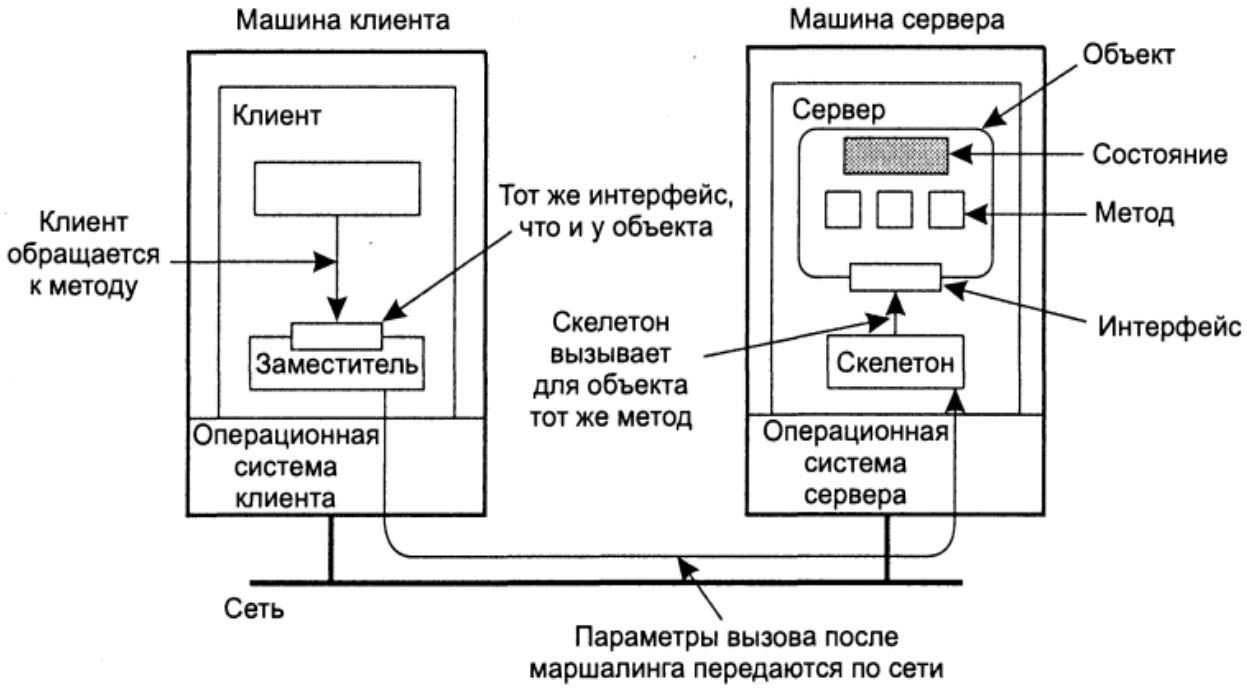

Рисунок 1.4 — Схема реализации удаленного объекта на машинах клиента и сервера [4]

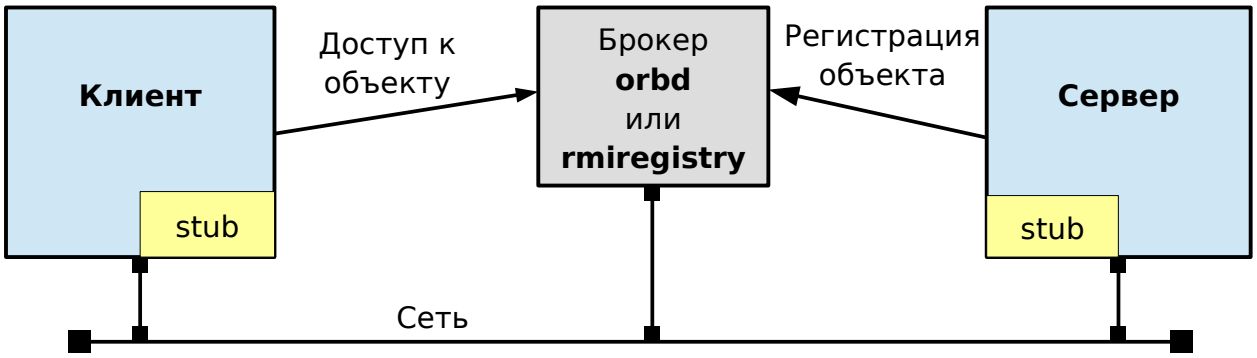

Рисунок 1.5 — Объектная модель взаимодействия «Клиент-сервер»

1.2 Становление систем с сервис-ориентированной архитектурой

К концу 90-х годов реализация распределенных приложений, интенсивно использующих технологии RPC, CORBA и RMI, породила различные проблемы их интеграции. Источник этих проблем кроется в самой технологии реализации основанной на том, что интерфейсы взаимодействия программ-клиентов и программ-серверов, хоть описываются на универсальном языке **IDL** (***Interface Definition Language***), но в силу генерации специальных программных заглушек (***stub***) делает модель «*Клиент-сервер*» сильносвязанной.

Эта сильная связанность приводит к следующим последствиям:

а) вся распределенная система должна создаваться одним разработчиком, который способен согласовать интерфейсы взаимодействия программ-клиентов и программ-серверов;

б) конкуренция среди разработчиков РВ-сетей заставляет их создавать функционал, который охватывает потребности максимально большего числа потенциальных заказчиков (покупателей) распределенных систем, делая эти системы большими, сложными и дорогими;

в) у заказчика (покупателя) распределенной системы имеются серьезные проблемы с ее адаптацией или модификацией приобретенной системы применительно к собственным потребностям, потому что он не имеет должной квалификации для успешного ее запуска и не знает в деталях как адекватно реализовать интерфейсы между программами-клиентами и программами-серверами;

г) разработчики РВ-сетей несут серьезные издержки по обеспечению гарантийных обязательств перед заказчиками;

д) заказчики (покупатели) имеют серьезные издержки по сопровождению купленной РВ-сети и ее развитию;

е) в сфере промышленного производства стала возникать «*островная*» или «*лоскутная*» автоматизация, препятствующая созданию комплексных систем автоматизации и принятию согласованных решений.

Перечисленные последствия привели к тому, что технологии РВ-сетей стали разделяться на два направления: *распределенные высокопроизводительные вычисления* и *распределенные сервис-ориентированные системы*.

Распределенные высокопроизводительные системы (**РВПС**) — технологическое направление, интегрирующее вычислительные ресурсы множества гетерогенных вычислительных систем, слабосвязанных между собой сетью ВМ. В процессе развития, за системами этого направления закрепился термин «***GRID-системы***», а за использующими их алгоритмами — термин «***GRID-вычисления***».

Распределенные сервис-ориентированные системы (**РСОС**) — технологическое направление, которое меняет *сильносвязанную* классическую модель «*Клиент-сервер*» (***strong coupling***) на *слабосвязанную* модель взаимодействия (***loose coupling***) между потребителем сервиса и поставщиком сервиса.

Потребитель сервиса — клиент (программа-клиент), осуществляющая обрашение к поставщику сервиса на основе опубликованного интерфейса сервиса (функции, метода и тому подобного действия серверного ПО).

Поставщик сервиса — сервер (программа-сервер), осуществляющая публикацию интерфейсов сервисов и доступ к их функциям или методам.

Сервис — зонтичный термин, обозначающий серверное ПО некоторой вычислительной системы, обеспечивающий функциональные потребности по-

требителя сервиса и реализующий необходимые вычисления в условиях слабого взаимодействия между потребителем и поставщиком сервиса.

Если технологии GRID-систем развивалось естественным путем в рамках достаточно больших и специализированных проектов, поддерживаемых группами высококлассных специалистов крупных вычислительных и научных центров, то направление РСОС вынуждено было разрабатывать и использовать технологии, ориентированные на широкий круг потребителей сервисов. Такая ситуация потребовала поиска и адаптации новых идей, не связанных напрямую с вычислительной парадигмой использования ВМ.

Начальный этап становления систем с новой сервис-ориентированной архитектурой следует рассмотреть на базе следующих вопросов:

1) развитие концепции и успехи реализации web-технологий;
2) формирование универсальной концепции SOA.

1.2.1 Развитие web-технологий

Началом становления web-технологий считается 1990 год, когда британский ученый Тимоти Джон Бернерс-Ли придумал термин «*Всемирная путина*» или *World Wide Web*.

Вспомнить базовые идеи web-технологий студент может из бакалаврского курса, изложенного в учебном пособии [1, подраздел 4].

В основу web-технологий положены три технологические новинки:

а) **URI** (*Uniform Resource Identifier*) — унифицированный идентификатор ресурса;

б) **HTML** (*HyperText Markup Language*) — гипертекстовый язык разметки;

в) **Браузер** (*web browser*) — прикладная программа для просмотра содержимого HTML-файлов, включающих файлы рисунков формата ****.gif***.

С позиции РСОС, web-технологии интересны прежде всего тем, что была представлена новая модель реализации архитектуры «*Клиент-сервер*», которая предполагает ***слабосвязанное взаимодействие*** двух групп программ: *Web-серверов* и *web-браузеров*.

Web-сервер — программное обеспечение, поддерживающее интерфейс ***CGI*** (*Common Gateway Interface*), что предполагает:

а) возможность подключения к нему по протоколу прикладного уровня HTTP и передачу запроса на ресурс с помощью URI;

б) возможность формирования ответа на запрос в виде HTML-страницы и

передачи ее автору запроса по протоколу HTTP.

в) **Web-браузер** — программное обеспечение, способное:

г) сформировантъ запрос на ресурс средствами синтаксиса языка URI;

д) соединиться с web-сервером по протоколу TCP и передать ему запрос на ресурс средствами протокола HTTP;

е) принять от сервера результат запроса в виде HTML-страницы;

ж) выполнить собственными средствами отображение HTML-станицы в виде, управляемом тегами языка HTML.

Получив HTML-страницу, web-браузер проводит разбор ее состаных частей и делает дополнительные запросы на дополнительные ресурсы, например, файлы рисунков.

По получению всех необходимых ресурсов, web-браузер инициирует разрыв соединения со всеми web-серверами.

После разрыва соединения web-сервер «забывает» о выполненном запросе и не фиксирует соответствующее состояние web-браузера.

В архитектурном плане плане, обе группы ПО (сервера и браузеры) не представляли какой-либо строго организованной системы. Они произвольно взаимодействовали в сети ***Интернет*** (*Internet*), опираясь на стек протоколов ***TCP/IP*** и используя средства ***DNS*** (*Domain Name System*) для разрешения проблем с адресацией. В таком виде web-технологии не представляли никакой конкуренции бурно развивающейся технологии CORBA и решали свои внутренние проблемы, специфичные для каждой группы ПО.

Программное обеспечение web-серверов сосредоточилось на качественной поддержке интерфейса CGI, протокола HTTP и различных версий, развивающих возможности языка HTML. В концептуальном плане возникает потребность в создании динамических HTML-страниц, обеспечивающих новый более интеллектуальный уровень доступа к ресурсам Интернет.

В июне 1995 года, датский программист ***Расмус Лердорф***, экспериментировавщий на языке ***Perl*** с CGI-шаблонами языка HTML, опубликовал описание языка ***PHP*** (*Personal Home Page Tools*) — Инструменты для создания персональных web-страниц. В дальнейшем, этот скриптовый язык стал называться «*Hypertext Preprocessor*» или - «*Препроцессор гипертекста*».

В начале 1995 года, организация-фонд ***Apache Software Foundation*** выпустила свободный web-сервер ***Apache HTTP Server***, написанный на языке C и расширяющий свои функциональные возможности посредством подключения различных модулей. Среди таких модулей, которых уже насчитывается более 500 единиц, одним из первых стал поддерживаться модуль языка PHP, а учитывая высокое быстродействие сервера, этот язык получил широкую популярность и стал своего рода эталонным представителем CGI-препроцессоров.

Что касается программного обеспечения web-браузеров, то развиваясь достаточно независимо, оно также претерпело значительные изменения.

В декабре 1995 года, американский программист Брендан Эйх (***Айк***) опубликовал скриптовый объектно-ориентированный язык программирования, названный ***JavaScript*** и предназначенный для браузера ***Netscape*** одноименной американской компании, работающей в сфере IT-индустрии.

В мае 1995 года, компания ***Sun Microsystems*** опубликовала первую версию языка Java, в котором присутсвовал пакет ***java.applet***, реализующий графические интерактивные возможности Java, выполняемые в среде Web-браузеров (правда с необходимостью запуска виртуальной машины Java — ***JVM***).

В августе 1996 года, корпорация ***Microsoft*** выпустила браузер ***Internet Explorer 3.0***, в котором в качестве скриптов был использован язык ***JScript***, синаксически основанный на языке ***JavaScript*** компании ***Netscape***. Это событие на долгие годы вызвало серьезные трения между этими компаниями.

В декабре 1996 года, ***CSS Working Group*** выпустила спецификации языка ***CSS*** (*Cascading Style Sheets*) — языка каскадной таблицы стилей, предначенной для описания внешнего вида документов. Это событие вызвало серьезную переработку программного обеспечения всех Web-браузеров.

В течение многих лет ПО браузеров адаптировалось и под различные версии языка HTML, включая XHTML, который соответствует специфика-циям метаязыка SGML. В настоящее время, стандартом является версия языка HTML 5.

В июне 1997 года, компания ***Sun Microsystems*** опубликовала новые пакеты ***javax.servlet*** и ***javax.servlet.http*** языка Java, которые были разработаны для платформы Java EE. Уже в 1999 году, компания-фонд ***Apache Software Foudation*** выпустила свободый сервер, являющийся контейнером сервлетов и JSP-страниц.

Таким образом, к 2000 году web-технологи полностью сформировались для реализации полноценных приложений, хотя бы на платформе языка Java.

1.2.2 Развитие концепции SOA

Недостатки, присущие объектному подходу создания РСОС, и все возрастающие потребности производства в управлении бизнес-процессами потребовали нового уровня абстракции, который был сформулирован как «*Сервисное Ориентирование*», показанное на рисунке 1.18 и влючающее [5]:

а) **BPM** (*Business process management*, **управление бизнес-процессами**) — концепция процессного управления организацией, рассматривающая

бизнес-процессы как ресурсы;

б) **EAI** (*Enterprise Application Integration*) — общее название сервиса интеграции прикладных систем предприятия;

в) **AOP** (A*spect-Oriented Programming*) — аспектно-ориентированное программирование; парадигма программирования, основанная на идее разделения функциональности программы на модули;

г) **Web-сервисы** (*Web-services*) — web-службы со стандартизированными интерфесами, адресуемые уникальными URI/URL-адресами.

Концептуально, на уровне сервисного-оринтированой архитектуры выделяются следующие составляющие:

а) **Сервисные компоненты** (или сервисы) описываются программными компонентами, которые обеспечивают прозрачную сетевую адресацию.

б) **Интерфейс сервиса** обеспечивает описание возможностей и качества предоставляемых сервисом услуг. В таком описании определяется формат сообщений для обмена информацией, а также входные и выходные параметры методов, поддерживаемых сервисным компонентом.

в) **Соединитель сервисов** — это транспорт, обеспечивающий обмен информацией между отдельными сервисными компонентами.

г) **Механизмы обнаружения сервисов** предназначены для поиска севисных компонентов, обеспечивающих требуемую функциональность сервиса.

Рисунок 1.6 — Концептуальная идея сервис-ориентированной архитектуры [5]

В 1998 году Дейв Винер, сотрудник компании ***UserLand Software***, опубликовал протокол ***XML-RPC***, предназначенный для вызова удаленных процедур в формате XML-сообщений поверх протокола HTTP. Сам протокол разрабатывался по заказу корпорации Microsoft для обеспечения решения задач в области электронной коммерции. Последующие доработки этого протокола привели к созданию протокола *SOAP* (*Simple Object Access Protocol*).

Известны две спецификации этого протокола, опубликованные консорциумом ***W3C***:

а) Simple Object Access Protocol (SOAP) 1.1 [6];

б) SOAP Version 1.2 Part 0: Primer (Second Edition) [7].

В марте 2001 года и в июне 2007 года, консорциум W3C публикует спецификации ***WSDL*** (*Web Service Description Language*) в документах [8] и [9]. Эти спецификации обеспечивают описание сервисов в формате XML-документов, что позволяет их публикацию средствами web-технологий.

Считается, что в августе 2000 года была реализована первая система ***UDDI*** (*Universal Description Discovery & Integration*) — инструмент для размещения описаний описаний WSDL. В настоящее время доступно описание версии ***UDDI 3.0.2*** [10], которое находится под контролем глобального консорциума ***OASIS*** (*Organization for the Advancement of Structured Information Standards*).

Таким образом, на начало 2000-х годов были сформированы все необходимые технологические компоненты для создания сервис-ориентированных систем на основе ***web-сервисов первого поколения***.

Такие системы имели общую архитектуру взаимодействия «*Клиент-сервер*», показанную на рисунке 1.7.

Рисунок 1.7 — Взаимодействие участников SOA для web-сервисов первого поколения [11]

К числу недостатков ***web-сервисов первого поколения*** традиционно относят отсутствие единых протоколов авторизации и аутентификации пользователей, что требовало от разработчиков систем самостоятельной разработки недостающего функционала.

Такая ситуация не способствовала стандартизации и унификации разрабатываемых программных продуктов. Устранение такой ситуации позволяет говорить о создании ***web-сервисов второго поколения***.

Для подробного обсуждения всей проблематики web-сервисов первого поколения, у нас пока — мало знаний. Вернемся к этому вопросу в пятой главе, когда более подробно познакомимся с сопутствующими технологиями программной платформы Java EE, а сейчас необходимо определиться с целевыми парадигмами, отвечающими на вопрос: «*Для каких глобальных целей или хотябы только их ориентиров необходмы сервис-ориентированные технологии*».

Среди множества возможных направлений исследования можно выделить вполне очевидную тематику — парадигмы сервис-ориентированных архитектур, рассмотренных в проекции автоматизации промышленных предприятий и других организаций.

1.3 Современные парадигмы сервис-ориентированных архитектур

Основная проблематика современных сервис-ориентированных архитектур — сложность их реализации и последующего сопровождения.

Концептуальная идея SOA не сводится только к реализации и взаимодействию web-сервисов, хотя такая парадигма является преобладающей. В общем случае считается, что в системе реализованной по парадигме SOA:

а) имеются ***сервисы***, которые создаются и сопровождаются некоторыми ***провайдерами сервисов***;

б) имеются ***потребители сервисов***, которые ищут и используют функциональные возможности нужных им сервисов;

в) имеются ***бизнес-процессы***, каждый из которых реализуется одним или несколькими потребителями сервисов.

Целевое назначение такой системы — успешная реализация всех бизнес-процессов потребителями сервисов, что порождает два проблемных вопроса:

а) ***в какой форме*** должен быть представлен сервис, чтобы потребитель сервиса мог его найти и успешно использовать в реализации целевого бизнес-процесса?

б) ***какими средствами*** необходимо реализовывать сервисы провайдеру, чтобы сервис был доступен потребителю в приемлемый срок?

В современных условиях развития информационных технологий, наиболее приемлемой формой ***для потребителя сервиса*** является web-сервис.

Современные браузеры представляют собой достаточно сложные и развитые инструментальные средства для доступа в сети Интернет, поиска web-сервисов и работы с ними. Этот факт снимает первый проблемный вопрос, обозначенный выше, и позволяет сосредоточить внимание на второй проблеме: реализации web-сервисов провайдерами сервисов.

В современных условиях развития информационных технологий, наиболее приемлемым инструментом ***для поставщика сервиса*** является сервер приложений, обеспечивающий средства разработки и сопровождения web-сервисов.

Данное утверждение не является столь очевидным как предыдущее, а отражает тот факт, что в современных условиях имеются уже готовые, свободные для распространения инструментальные средства, которые любой поставщик сервиса может равернуть на своей рабочей станции или в сети для обеспечения индивидуальной или групповой разработки сервисов. В последующем, результаты такой разработки могут быть развернуты для эксплуатации и сопровождения результатов своей работы.

Хотя приведнная аргументация может показаться слишком ограниченной, она вполне приемлема для ограниченных условий учебного процесса.

Учебная цель данного подраздела — изучение теоретических положений и практических рекомендаций, которые используют провайдеры сервисов в плане реализации систем, поддерживающих парадигмы SOA.

Для достижения заявленной цели, в следующих четырех пунктах данного подраздела даются наиболее обощенные представления о концепции SOA:

а) ***эталонная модель SOA*** представляет наиболее абстрактную теоретическую часть используемой парадигмы, обеспечивая студента наборов терминов всей предметной области дисциплины;

б) ***модель Захмана*** дает наиболее абстрактную модель предприятия, которая сужает теоретические представления модели SOA, раскрывая 36 технологических позиций внимания разработчика сервисных систем;

в) ***концепция среды открытой системы***, известная как эталонная модель СОС POSIX; она представляет рекомендации по стандартизации, на кото-

рые должен ориентироваться любой разработчик информационных систем;

г) ***бизнес-парадигма модели SOA***, конкретизирующая эталонную модель SOA посредством смещения целевой установки архитектуры в пользу бизнес-руководителей предприятия.

Последующие два подраздела данной главы завершают общее описание предметной области изучаемой дисциплины, рассматривая:

а) ***программную платформу Java Enterprise Edition***, конкретизирующую теоретический профиль используемых инструментальных средств разработки SOA-систем;

б) ***инструментальные средства реализации PCOC***, конкретизирующие практический профиль разработки, развертывания и сопровождения целевых SOA-систем.

1.3.1 Эталонная модель SOA

Эталонная модель SOA была опубликована глобальным консорциумом OASIS в августе 2006 года [12]. Центральное место в ней занимает понятие сервиса (службы, услуги).

Service - это механизм, обеспечивающий доступ к одной или нескольким функциям, когда доступ предоставляется с использованием предписанного интерфейса и осуществляется в соответствии с ограничениями и политиками, указанными в описании службы. Услуга предоставляется субъектом — ***поставщиком услуг*** — для использования другими, но конечные ***потребители услуги*** могут быть неизвестны поставщику услуг и могут демонстрировать использование услуги за пределами объема, первоначально задуманного поставщиком.

Доступ к услуге осуществляется через ***интерфейс службы***, где интерфейс содержит особенности доступа к базовым возможностям сервиса. Нет никаких ограничений на то, что составляет основную возможность сервиса или как доступ реализован поставщиком услуг.

Таким образом, служба (сервис) может выполнять свою описанную функциональность посредством одного или нескольких автоматизированных и/или ручных процессов, которые сами могут вызывать другие доступные службы.

Услуга непрозрачна в том смысле, что ее реализация обычно скрыта от потребителя службы, за исключением, когда:

а) информация и поведение моделей предоставляются через интерфейс службы;

б) потребителям сервиса необходимо определить (оценить) необходимость использования услуги.

Эталонная модель SOA определяет три фундаментальных понятия, которые важны для понимания того, что вовлечено во взаимодействие со службами: видимость (visibility) между поставщиками и потребителями услуг, взаимодействие (interaction) между ними и реальный эффект взаимодействия со службой (real world effect).

Visibility — требование, чтобы поставщик услуг и потребитель взаимодействовали друг с другом, они должны иметь возможность «видеть» друг друга. Это верно для любых отношений между потребителем и поставщиком - в том числе в прикладной программе, где одна программа вызывает другую: без наличия соответствующих библиотек вызов функции не может быть завершен. В случае SOA необходимо подчеркнуть видимость, поскольку не обязательно очевидно, как участники службы могут видеть друг друга.

Interaction — деятельность, связанная с использованием предлагаемой возможности сервиса, направленная на достижение конкретного желаемого реального эффекта.

Real world effect — фактический результат использования услуги, а не просто возможность, предлагаемая поставщиком услуг.

Представленные четыре понятия создают статическую часть теоретической концепции SOA, представленную рисунком 1.8.

Концептуальные отношения между предсталенными понятиями требуют введения дополнительных трех понятий.

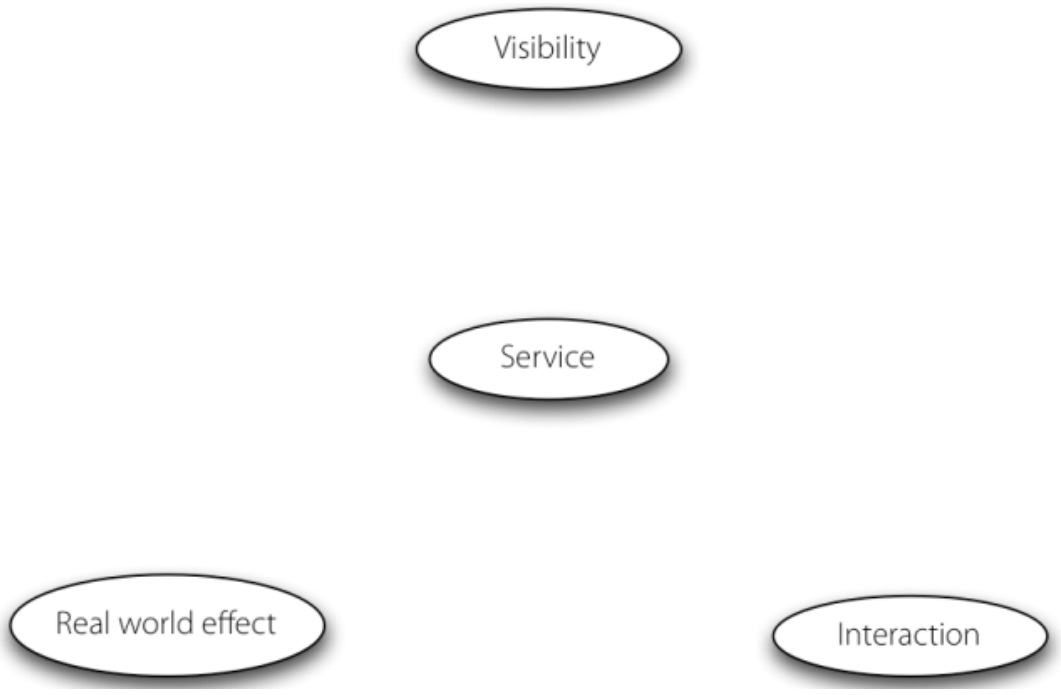

Рисунок 1.8 — Базовые концепции вокруг понятия сервиса [12]

Execution context — набор технических и бизнес-элементов, которые об-

разуют путь между теми, у кого есть потребности и у кого есть возможности, позволяющие поставщикам и потребителям услуг взаимодействовать.

Contract&Policy — ***Политика*** (policy) представляет собой некоторое ограничение или условие использования, развертывания или описания принадлежащего объекта, как это определено любым участником взаимодействия, ***Контракт*** (contract) представляет собой соглашение двух или более взаимодействующих сторон. Как и политики, соглашения (контракты) также касаются условий использования услуги; они также могут ограничивать ожидаемые результаты использования сервиса в реальном мире. Эталонная модель ориентирована в первую очередь на концепцию политики и контрактов применительно к услугам.

Service description (описание сервиса) — отличительный признак сервис-ориентированной архитектуры, касающийся большого количества сопутствующей документации и описания. Описание сервиса представляет информацию, необходимую для использования сервиса. В большинстве случаев не существует единого «правильного» описания, а требуемые элементы описания зависят от контекста и потребностей сторон, использующих ассоциированный объект. Несмотря на то, что существуют определенные элементы, которые могут быть частью описания любой услуги, в частности, информационная модель, многие элементы, такие как функция и политика, могут различаться.

С учетом дополнительных трех понятий, предметная область концептуальной части SOA представляется рисунком 1.9.

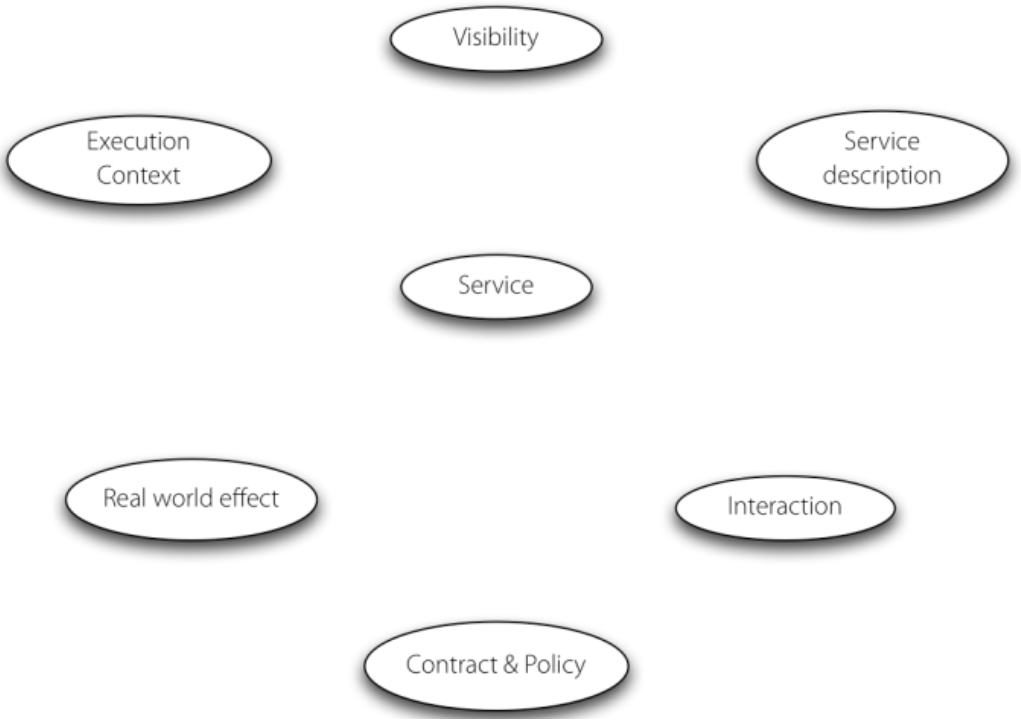

Рисунок 1.9 - Основные понятия в базовой модели SOA [12]

Что касается назначения и значимости самой модели, то они раскрываются следующими определениями [12].

Reference Architecture (Эталонная архитектура) - это шаблон архитектурного проектирования, который указывает, как абстрактный набор механизмов и отношений, реализующих заранее определенный набор требований.

Reference Model (Эталонная модель) - это абстрактная структура для понимания значимых связей между объектами некоторой среды, которая позволяет разрабатывать конкретные архитектуры с использованием согласованных стандартов или спецификаций, поддерживающих эту среду. Она состоит из минимального набора объединяющих понятий, аксиом и отношений в конкретной проблемной области и не зависит от конкретных стандартов, технологий, реализаций или других конкретных деталей.

Service Oriented Architecture (Сервис-ориентированная архитектура, SOA) - это парадигма для организации и использования распределенных возможностей, которые могут находиться под контролем различных доменов собственности. Она предоставляет единые средства для предложения, обнаружения, взаимодействия и использования возможностей для получения желаемых результатов, соответствующих измеримым предварительным условиям и ожиданиям.

Дополнительные определения, входящие в эталонную модель, представлены в таблице 1.1.

Таблица 1.1 — Дополнительный глоссарий эталонной модели SOA [12]

Понятие	***Определение***
Action Model	Характеристика допустимых действий, которые могут быть вызваны против службы.
Awareness	Состояние, в котором одна сторона знает о существовании другой стороны. Осведомленность не подразумевает готовность или достижимость.
Behavior Model	Характеристика (и ответы, и временные зависимости между ними) действий над службой.
Capability	Реальный эффект, который поставщик услуг может предоставить потребителю.
Framework	Набор предположений, концепций, ценностей и практик, которые представляют собой способ просмотра текущей среды.
Idempotency/ Idempotent	Характеристика услуги, при которой множественные попытки изменить состояние всегда будут генерировать одно изменение состояния, только если операция уже была успешно завершена один раз.

Понятие	*Определение*
Information model	Характеристика информации, которая связана с использованием службы.
Offer	Приглашение использовать возможности, предоставляемые поставщиком услуг в соответствии с определенным набором политик.
Process Model	Характеристика временных отношений между временными свойствами действий и событий, связанных с взаимодействием с сервисом.
Reachability	Достижимость — способность потребителя и поставщика услуг взаимодействовать. Достижимость является аспектом видимости.
Semantics	Концептуализация подразумеваемого значения информации, которая требует слов и/или символов в контексте использования.
Semantic Engagement	Отношения между агентом и набором информации, которые зависят от конкретной интерпретации информации.
Service Consumer	Сущность, которая стремится удовлетворить конкретную потребность с помощью возможностей использования, предлагаемых посредством услуги.
Service Interface	Средства, с помощью которых осуществляется доступ к основным возможностям службы.
Service Provider	Субъект (физическое или юридическое лицо), который предлагает использование возможностей посредством услуги.
Shared state	Множество фактов и обязательств, которые проявляются для участников сервиса в результате взаимодействия со службой.
Software Architecture	Структура или структуры информационной системы, состоящей из сущностей и их внешне видимых свойств, а также отношений между ними.
Willingness	Предрасположенность поставщиков услуг и потребителей к взаимодействию.

В целом, эталонная модель не дает конкретных схем реализации и не привязана к каким-либо организационным структурам.

Концептуальная ценность эталонной модели SOA — расширение теоретических представлений конкретных проектировщиков и разработчиков сервис-ориентированных систем, указывая им на многоплановость возможных проектных решений и последующих реализаций.

Непосредственная конкретизация модели SOA предполагает введение дополнительных ограничений на предметную область ее использования.

1.3.2 Модель Захмана

В 1987 году, Джон Захман (сотрудник корпорации IBM) предложил иерархическую модель предприятия, предлагая на каждом уровне представления давать ответы на вопросы: ***что?***, ***как?***, ***где?***, ***кто?***, ***когда?*** и ***почему?***

Предложенная модель множество раз модифицировалась и расширялась. В конечном итоге она приняла стабильную форму и стала обязательным атрибутом учебных пособий по архитектуре предприятий, например, [13, 14]. Общий вид ее представлен на рисунке 1.10.

		Данные ЧТО	Функции КАК	Дислокация, сеть ГДЕ	Люди КТО	Время КОГДА	Мотивация ПОЧЕМУ	
Бизнес-руководители	*Планировщик*	Список важных понятий и объектов	Список основных бизнес-процессов	Территориальное расположение	Ключевые организации	Важнейшие события	Бизнес-цели и стратегии	Сфера действия (контекст)
	Владелец, менеджер	Концептуальная модель данных	Модель бизнес-процессов	Схема логистики	Модель потока работ (workflow)	Мастер-план реализации	Бизнес-план	Модель предприятия
ИТ-менеджеры и разработчики	*Конструктор, архитектор*	Логические модели данных	Архитектура приложений	Модель распределенной архитектуры	Архитектура интерфейса пользователя	Структура процессов	Роли и модели бизнес-правил	Модель системы
	Проектировщик	Физическая модель данных	Системный проект	Технологич. архитектура	Архитектура презентации	Структуры управления	Описания бизнес-правил	Технологическая (физическая) модель
	Разработчик	Описание структуры данных	Программный код	Сетевая архитектура	Архитектура безопасности	Определение временных привязок	Реализация бизнес-логики	Детали реализации
		Данные	Работающие программы	Сеть	Реальные люди, организации	Бизнес-события	Работающие бизнес-стратегии	Работающее предприятие
		Данные	Функции, Процессы	Сеть, расположение систем	Люди, организации	Время, расписания	Мотивация	

Рисунок 1.10 — Архитектурная матрица модели Захмана [14]

Для нашей дисциплины познавательная ценность модели Захмана состоит в демонстрации сложностей, с которыми сталкиаются участники процессов масштабной автоматизации деятельности предприятий.

Позитивная сторона модели Захмана связана с наглядным представлением ***тридцати шести позиций*** внимания, требующих согласования между представителями бизнеса (руководства) и техническими представителями ограниченных организационных структур. Это позволяет им контролировать полноту принятых проектных решений.

Негативная сторона модели Захмана связана ***с качественным различием*** компетенций и ответственности между бизнес-руководителями и ИТ-специалистами. Это перегружает каждую из указанных групп большим количеством малопонятных модельных представлений.

1.3.3 Концепция среды открытой системы

В 1998 году, американский Институт инженеров электротехники и электроники (*Institute of Electrical and Electronics Engineers, IEEE*) выпустил стандарт **IEEE Std 1003.23—98** под названием «Стандарт института инженеров по электротехнике и электронике. Руководство по проектированию профилей среды открытой системы организации-пользователя». Он исходит из того, что современная инфраструктура предприятия представляет собой гетерогенную программно-аппаратную среду, использующую компьютеры различных классов и производителей. Стандарт предлагает формировать среду открытой системы (СОС), которая бы решала проблему совместимости трех инфраструктурных ресурсов: информационных, вычислительных и телекоммуникационных.

В ноябре 2002 года, Госстандарт России вынес постановление о публикации Рекомендаций по стандартизации **Р 50.1.041-2002** под названием «Руководство по проектированию профилей среды открытой системы (СОС) организации-пользователя» [15], где указанная выше концепция представлена как «Эталонная модель СОС POSIX (ИСО/МЭК 14252)».

Идейная основа эталонной модели СОС POSIX состоит в том, что при должном формировании профилей СОС (***OSE***, ***Open System Environment***) предприятий, дальнейшее решение осуществляется рынком постащиков программного обеспечения, аппаратных средств и телекоммуникаций.

Про своей сути, Рекомендации по стадартизации Р 50.1.041-2002 [15] являются описанием методики проектирования информационных систем, где на верхнем (концептуальном) уровне выделяются и иерархически размещаются три объектных сущности и два интерфейса:

а) **Объект прикладных программных средств**;

б) **ППИ** — прикладной программный интерфейс (***API, Application Program Interface***);

в) **Объект прикладной платформы** — объект набора ресурсов, включая

технические и программные средства, которые обеспечивает услуги для функционирования прикладных программных средств;

г) **ИВС** — интерфейс с внешней средой (***EEI, External Environment Interface***);

д) **Внешняя среда**.

Выделенные объектные сущности объединяются с помощью четырех служб, как это показано на рисунке 1.11.

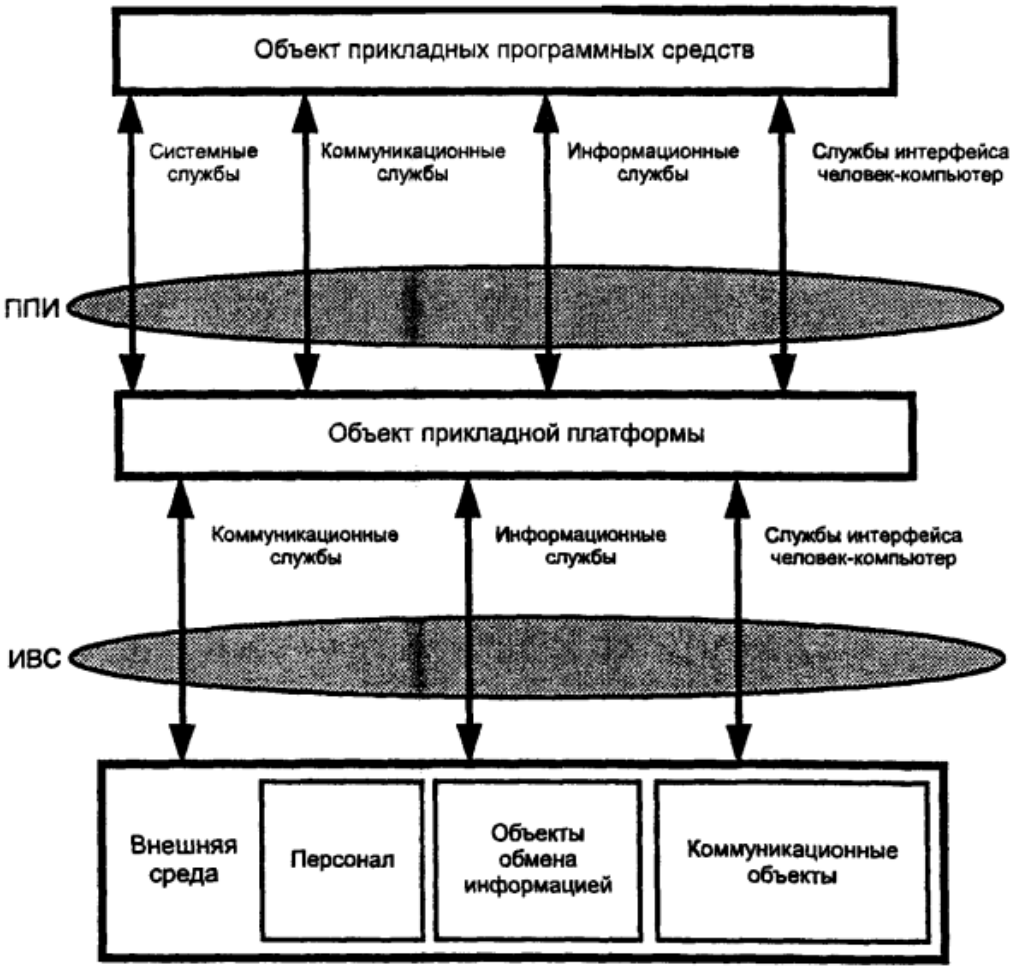

Рисунок 1.11 — Эталонная модель СОС POSIX [15]

Далее, следуя описанию методики проектирования, проводится декомпозиция объектных уровней и интерфейсов. Выполняются различные группировки по службам и технологическим компонентам, строятся матрицы перекрестных связей.

Позитивная сторона модели СОС POSIX — сведение всего многообразия сервисов к четырем службам: системной, коммуникационной, информационной и службы интерфейса человек-компьютер. Это усиливает понимание

между ИТ-специалистами и Бизнес-руководителями предприятий, поскольку предоставляет последним модельные представления о хозяйственной деятельности предприятия.

Негативная сторона модели COC POSIX — явные ограничения на число и состав выделенных служб (сервисов). Поскольку ИТ-службы являются обеспечивающими подразделениями по отношению к производственным подразделениям предприятия, то возникают проблемы взаимной подчиненности таких служб и степени их ответственности, по отношению к производственной деятельности предприятия.

1.3.4 Бизнес-парадигма модели SOA

Бизнес-парадигма модели SOA исходит из того, что сервисы существуют исключительно для удовлетворения бизнес-руководителей предприятия. Тогда сама сервисная архитектура может быть представлена трехуровневой иерархией, как это показано на рисунке 1.12.

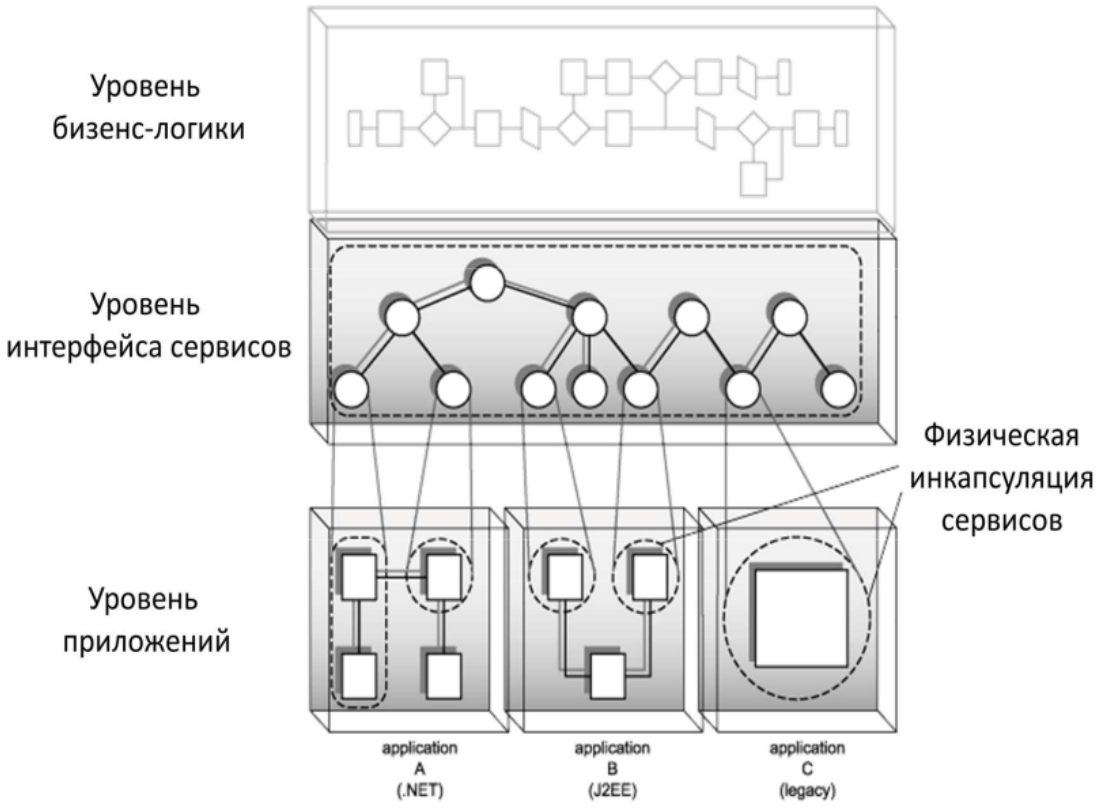

Рисунок 1.12 — Бизнес-парадигма архитектуры предприятия [11]

Уровень бизнес-логики — соответствует уровню бизнес-руководителей предприятия, которые генерируют бизнес-процессы и становятся потребителями сервисов (***Service Consumer***).

Уровень интерфейса сервисов (***Service Interface***) — уровень взаимодей-

ствия бизнес-руководителей и ИТ-специалистов предприятия, обеспечивающий свойства видимости (***Visibility***), перспектива возможного использования сервисов (***Interaction***) и фактического получения результата услуги (***Real world effect***). На этом уровне ИТ-специалисты получают статус провайдеров услуг (***Service Provider***).

Уровень приложений — уровень ИТ-специалистов, которые обеспечивают реализацию сервисов. Он считается «невидимым» для бизнес-руководителей, поэтому может реализовываться различными способами, например:

а) представлять слабосвязанные или сильносвязанные системы;

б) быть распределенными или сосредоточенными системами;

в) соответствовать или несоответсвовать эталонной модели СОС POSIX;

г) принадлежать предприятию или находиться в «облаке» (*cloud computing*) на аутсорсинге (*outsourcing*).

Позитивная сторона модели бизнес-парадигмы модели SOA состоит в разделении статуса и полномочий ИТ-специалистов и Бизнес-руководителей предприятий. В этой модели наблюдается возможность масштабирования и смены парадигм уровня приложений без прямого влияния на уровень бизнес-логики предприятия. Отсутствуют многоплановые матричные зависимости, присутствующие в предыдущих двух концепциях.

Ситуация принципиально не меняется, когда основная производственная деятельность предприятия связана с ИТ-технологиями. В этом случае, основная часть бизнес руководителей является или являлась ранее ИТ-специалистами. Тогда в силу вступает иерархия по целям производственной деятельности: стратегия, тактика и оперативное исполнение. Этого вполне достаточно для сохранения целостности модельной парадигмы SOA.

Негативная сторона модели бизнес-парадигмы модели SOA состоит в достаточно высокой степени неопределенности концепции проектирования и реализации уровня приложений.

Бизнес-парадигма модели SOA предполагает быстрое и качественное создание, развертывание и сопровождение сервисов.

Бизнес-руководители — главные потребители сервиса не могут удовлетвориться медленным созданием и развертыванием нужных сервисов. Такое ограничение стимулирует качественный выбор прикладной платформы.

1.4 Программная платформа Java Enterprise Edition

Проблема качественного выбора прикладной платформы упирается в решение двух подпроблем (см. рисунок 1.12):

- выбор платформы, обеспечивающий ***стабильное представление*** и легко расширяемую реализацию уровня интерфейсов сервисов;
- выбор платформы, обеспечивающей реализацию ***стабильных прикладных компонент*** уровня приложений.

23 мая 1995 года публичная американская компания Sun Microsystems выпустила первую версию программной платформы, включающей:

- язык объектно-ориентированного программирования Java;
- виртуальную машину JVM (*Java Virtual Machine*).

Позитивная часть этой реализации — обеспечение разработчиков программного обеспечения уровня приложений инструментальными средствами для создания программных компонент, переносимых между аппаратными платформами и платформами операционных систем.

В 1998 году таже компания Sun Microsystems анонсировала платформу **JPE** (*Java Platform for the Enterprise*), которая содержала спецификацию технологии написания и поддержки серверных компонентов реализации бизнес-логики обозначаемой как **EJB** (*Enterprise Java Beans*). В дальнейшем, реализации этой платформы стали обозначаться как продукты Java EE.

Позитивная составляющая концепции этой платформы — контейнерная технология реализации сервисов, что позволяет унифицировать как реализацию компонент уровня приложений, так и представления уровня сервисов.

Текущее состояние парадигмы Java Enterprise Edition — версия 8.

С момента появления первой реализации Java EE (декабрь 1999 года) прошло много времени, в течении которого технологии JPE постоянно модифицировались и дополнялись сопуствующими технологиями. В настоящее время, текущей и финальной реализацией Java EE является версия 8.0, появившаяся в сентябре 2017 года.

В 2018 году, правоприемница технологий Java и Java EE (корпорация Oracle) передала в руки некоммерческой организации Eclipse Foundation процессы разработки спецификаций, TCK (*Technology Compatibility Kit*) и эталонную реализацию продукта, но потребовала смены бренда Java EE и главного домена пакета — «javax.*».

На основании сложившейся ситуации, Eclipse Foundation сделала следующее:

а) заявила и представила новую платформу **Jakarta EE 8**;

б) заявила о переходе на новое пространство имен «**jakarta.***»;

в) сформировала новый процесс **JESP** (*Jackarta Eclipse Specification Process*) — формальный процесс, позволяющий заинтересованным лицам участвовать в формировании будущих версий спецификаций платформ;

г) сформировала рабочую группу **Jakarta EE Working Group**, которая будет развивать платформу Jakarta EE.

Учебная цель данного подраздела — изучение общей архитектуры приложений, основанных на принципах парадигмы Java Enterprise Edition.

Общие принципы парадигмы Java Enterprise Edition достаточно подробно изложены в современной литературе:

а) общие вопросы и проблематика использования платформы Java EE 8 подробно изложены в монографии Себастьяна Дашнера «Изучаем Java EE. Современное программирование для больших предприятий» [16];

б) фактический учебный материал данного пособия опирается на монографию Энтони Гонсалвеса «Изучаем Java EE 7» [17];

в) частные вопросы применения языка Java опираются на монографию Герберта Шилдта «Java 8. Полное руководство» [18].

Чтобы разобраться с этими принципами Java EE, воспользуемся материалом монографии [17] и рассмотрим следующие вопросы:

а) контейнеры и компоненты Java EE;

б) служебные сервисы контейнеров;

в) артефакты контейнеров;

г) аннотации и дескрипторы развертывания компонент;

д) управляемые компоненты платформы Java EE.

1.4.1 Контейнеры и компоненты Java EE

Основу архитектуры платформы Java EE составляют контейнеры и компоненты, которые написаны на языке Java и функционируют в среде виртуальной машины Java (JVM).

Контейнеры — специальные системные программы языка Java, создающие среду и сервис для функционирования других программных модулей языка Java, называемых ***компонентами***.

Компоненты — специальные расширяемые разработчиками модули языка Java для реализации сервисов, способные работать в среде контейнеров и

пользоваться его служебными сервисами.

Платформа Java EE имеет четыре вида стандартных контейнеров, представленных на рисунке 1.13:

а) ***EJB-контейнер*** — серверный вид контейнера, способный содержать только один вид компонента — «*EJB*»;

б) ***Веб-контейнер*** — серверный вид контейнера, способный содержать три вида компонент: «*EJB Lite*», «*Servlet (Сервлет)*» и «*JSF*»;

в) ***контейнер апплета*** — клиентский вид контейнера, реализуемый в среде браузера (при поддержке JVM) и способный содержать только один вид компонента — «*Applet (Апплет)*»;

г) ***контейнер клиентского приложения***, способный содержать только один вид компонента — «*Application (Приложение)*».

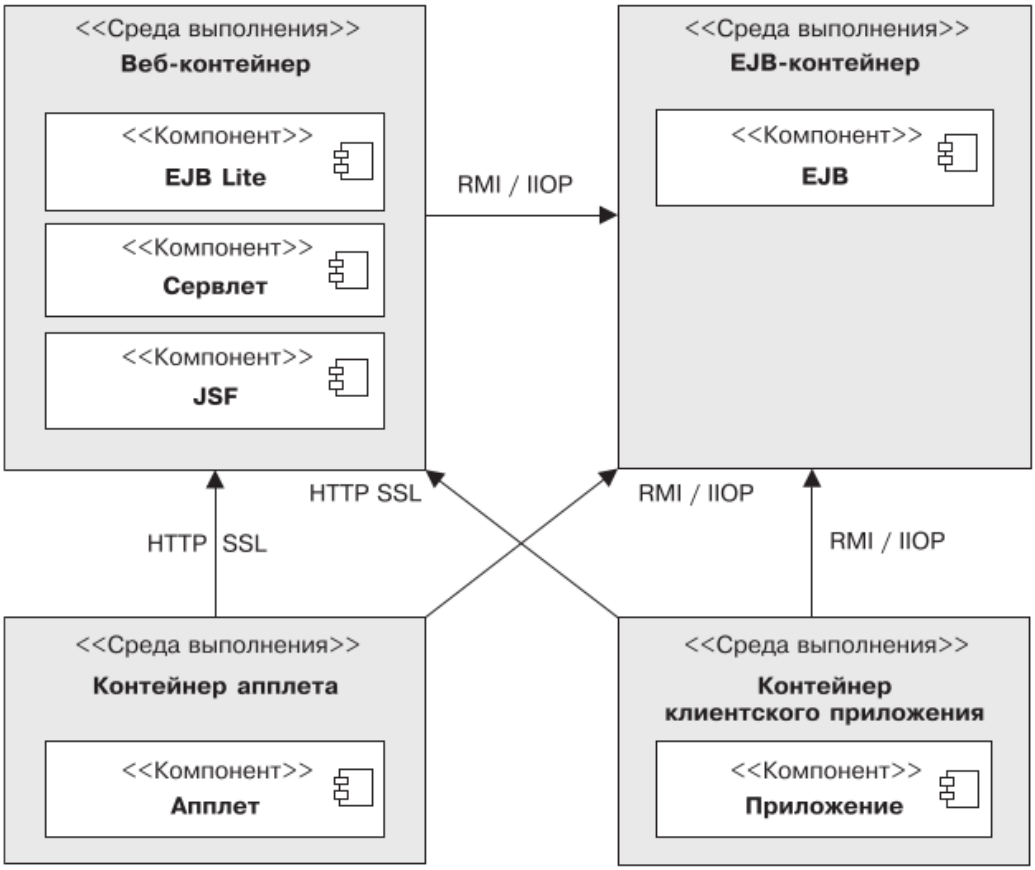

Рисунок 1.13 - Стандартные контейнеры Java EE [17]

Сразу же заметим, что контейнер апплетов не получил большой популярности, в основном по причине противодействия разработчиков браузеров, которые не желали поддерживать виртуальную машину Java (JVM). Поэтому он далее не рассматривается.

Обозначение **RMI/IIOP** читается как «RMI поверх IIOP».

RMI (*Remote Method Invocation*) — программный интерфейс для удаленного вызова методов на языке Java. Используется в реализациях объектного подхода в распределенных системах.

IIOP (*Internet Inter-ORB Protocol*) — конкретная реализация протокола GIOP на стеке протоколов TCP/IP.

GIOP (*General Inter-ORB Protocol*) — абстрактный протокол в распределенных объектных системах.

Основу платформы Java EE составляют контейнеры EJB и клиентских приложений.

EJB-контейнеры, именуемые серверами приложений, предназначены для реализации бизнес-логики крупномасштабных сервисов, а контейнеры клиентских приложений предназначены для реализации АРМ-пользователей, обеспечивающих доступ к серверам приложений. Выгоды от такой архитектуры информационных систем заключаются в использовании служебных сервисов предоставляемых контейнерами. Это освобождает разработчиков приложений от необходимости самостоятельного проектирования и реализации множества стандартных системных функций, например, синхронизации доступа к разделяемым ресурсам приложений, обеспечение безопастности и других задач.

Современные разработчики сервисов все больше ориентируются на использование веб-контейнеров.

Крупномасштабные EJB-компоненты требуют большого времени на проектирование, разработку и модификацию. В тех случаях, когда на уровне бизнес-логики (см. рисунок 1.12) масштабные бизнес-процессы представляются как комбинации более мелких бизнес процессов, гораздо эффективней использовать компоненты веб-контейнера (web-container).

Веб-контейнер обеспечивает доступ к нему по широко известным и популярным протоколам HTTP/HTTPS. Кроме контейнеров клиентских приложений можно использовать обычные браузеры, что обеспечивается тремя стандартными компонентами:

а) **EJB Light** — облегченный вариант EJB-контейнера, обеспечивающий наиболее важные функции реализации бизнес-логики и способный функционировать в веб-контейнере;

б) **Сервлет** (***HttpServlet***) — компонента, хорошо известная студентам по бакалаврскому курсу «*Распределенные вычислительные системы*» и поддерживающая технологию JSP-страниц;

в) **JSF** (***JavaServer Faces***) — компонента, обеспечивающая построение пользовательских интерфейсов с помощью XHTML-страниц на стороне серве-

ра (*подробно рассматривается во второй главе данного пособия*).

1.4.2 Служебные сервисы контейнеров

Контейнеры управляют расположенными в них компонентами, но предоставляют им служебные сервисы, как это показано на рисунке 1.14.

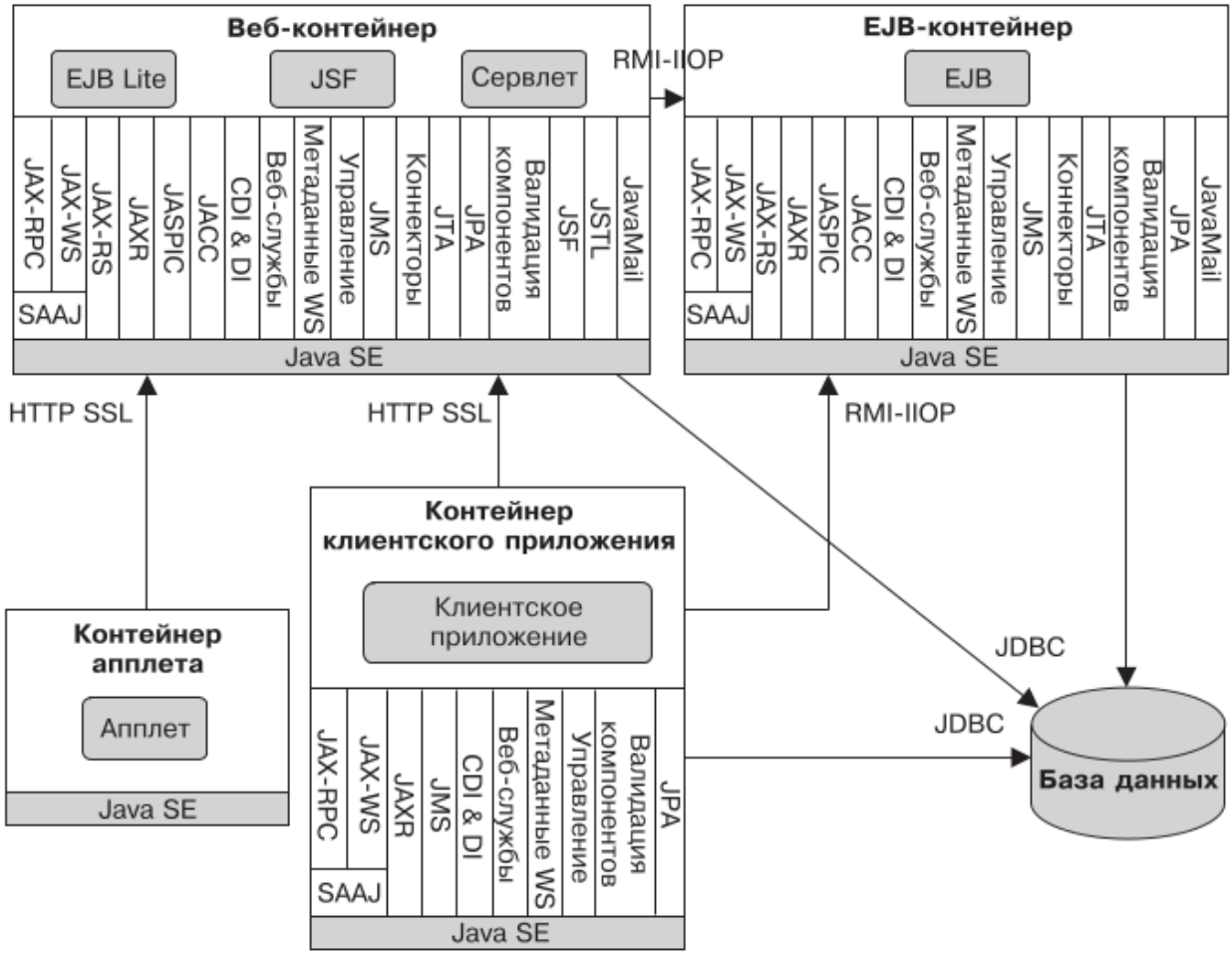

Рисунок 1.14 - Сервисы, предоставляемые контейнерами Java EE [17]

Основное назначение служебных (технических) сервисов — общее управление компонентами контейнеров, освобождая разработчиков сервисов от рутинной работы по их релизации. Часть сервисов имеют общее прикладное значение. Они обсуждаются далее в процессе изложения учебного материала.

Другая часть имеет важное прикладное значение для реализации бизнес-логики приложений. Им посвящены отдельные главы данного пособия:

а) **JSF** — сервис JavaSever Faces, обеспечивающий построение графических представлений веб-приложений, подробно обсуждается во второй главе;

б) **JPA** — сервис Java Persistence API ослуживает задачи обработки и хране-

ния данных, рассматривается в третьей главе;

в) **JAX-WS** — сервис, обеспечивающий функционирование веб-служб SOAP, излагается в пятой главе;

г) **JAX-RS** — сервис, обеспечивающий функционирование веб-служб в стиле REST, изложен в шестой главе.

1.4.3 Артефакты контейнеров

Разработанные компоненты, для последующего помещения в контейнер, необходимо стандартным образом упаковать в артефакты.

Результаты разработки любых программных проектов на языке Java всегда имеют некоторую структуру, состоящую из дерева каталогов и размещенных в них файлов. Для того, чтобы эти программы или библиотеки можно было бы использовать в других проектах их помещают в архивы, представляющие собой файлы формата ***ZIP*** с расширением ****.jar***. Дополнительным условием является:

а) наличие в корне архива каталога ***META-INF***;

б) наличие в этом каталоге файла манифеста с именем ***MANIFEST.MF***.

Подобные требования существуют и для результатов разработки компонент платформы Java EE, как это показано на рисунке 1.15.

Почти каждый артефакт контейнера имеет свой дескриптор развертывания.

Дескриптор развертывания — файл формата XML, описывающий каким образом компонент, модуль или приложение должны быть сконфигурированы.

Согласно рисунку 1.15, артефакт апплета имеет расширение ****.jar***, но не имеет дескриптора развертывания. Зато его архив должен быть подписан, а подпись зарегистрирована, иначе он не запустится.

Артефакты других контейнеров имеют следующие дескрипторы развертывания:

а) артефакты контейнера клиентского приложения имеют расширение ****.jar*** и дескриптор развертывания ***application-client.xml***, помещенный в каталог ***META-INF***;

б) артефакты веб-контейнера имеют:

1) расширение ****.war*** и дескриптор развертывания ***web.xml***, помещенный в каталог ***WEB-INF***;

2) если артефакт содержит и компоненту EJB Light, то дополнительно до-

бавляется дескриптор развертывания ***ejb-jar.xml***, помещенный в каталог ***WEB-INF***;

3) файлы JAVA-классов помещаются в каталог ***WEB-INF/classes***;
4) дополнительные JAR-файлы помещаются в каталог ***WEB-INF/lib***;

в) EJB-контейнер содержит артефакты двух видов:

1) артефакты компонент сеансов или компонент управляемых сообщениями, помещенные в JAR-архив, содержат дескриптор развертывания ***META-INF/ejb-jar.xml***;
2) корпоративные артефакты с расширением ****.ear*** имеют общий дескриптор развертывания ***META-INF/application.xml***.

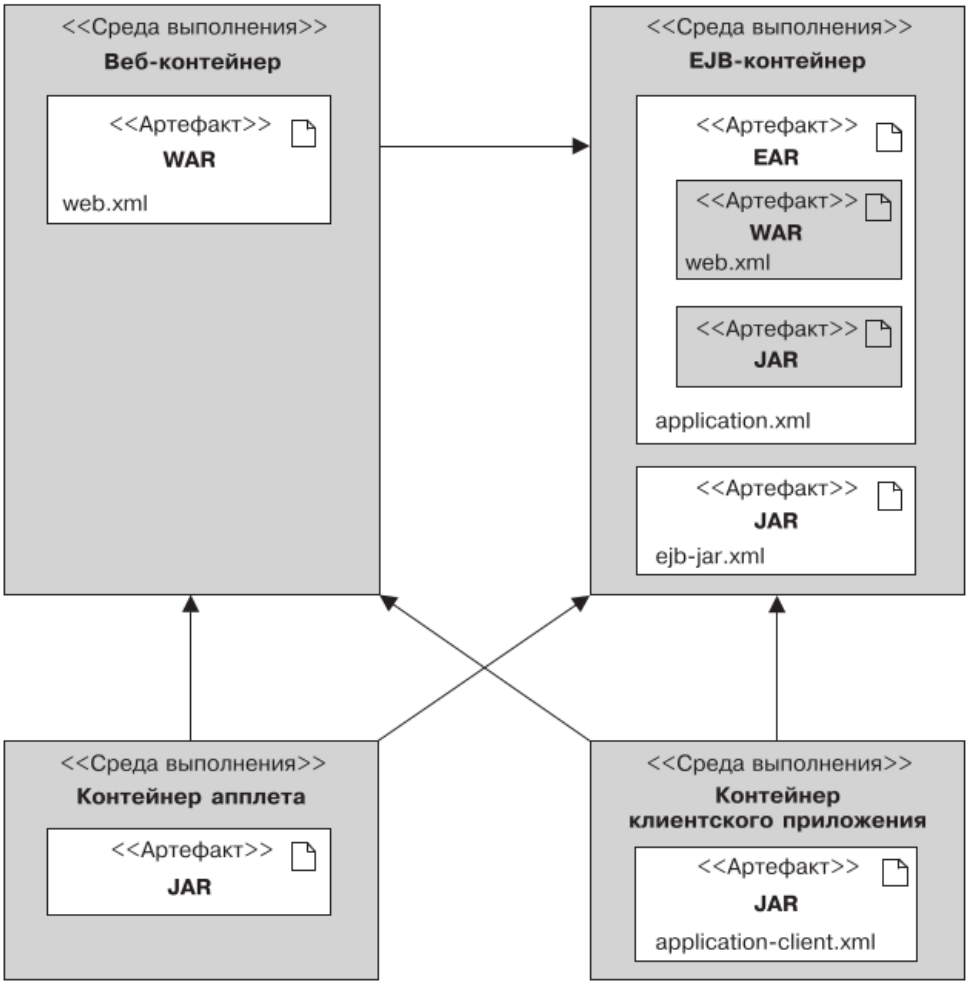

Рисунок 1.15 — Артефакты контейнеров Java EE [17]

Обычно создание артефактов обеспечивается инструментальными системами разработки.

Например, **Apache Maven** — фреймворк, автоматизирующий процесс сборки проектов платформы Java EE, имеет свой дескриптор развертывания ***pom.xml***.

1.4.4 Аннотации и дескрипторы развертывания

XML-файлы являются основой представления информации платформы Java EE.

Программная платформа Java EE широко использует файлы формата XML. Они применяются для представления различных данных, сериализации классов, содержимого передаваемых сообщений и других целей. Более подробно часть этих вопросов изложена в главе 4, а в данном пункте мы рассмотрим проблематику использования дескрипторов развертывания различных компонент.

Формальной основой развития программной платформы Java являются документы JSR, выпускаемые под различными номерами процессом Java Community Process (*JCP*). Этой же основы придерживается и программная платформа Java EE.

JSR (*Java Specification Request*) — «*Запрос на Спецификацию Java*», представляющий собой документ, описывающий спецификации и технологии, которые предлагается добавить к платформе Java.

В 1999 году, программная платформа Java EE содержала 10 документов JSR и включала компонент EJB, ориентированный на использование объектного подхода к реализации распределенных систем.

На момент выхода Java EE 7 (второй квартал 2013 года) число спецификаций JSR увеличилось до 31 и многие из них предлагали новые дескрипторы развертывания. Таблица 1.2 демонстрирует перечень спецификаций и место расположения их десрипторов в артефактах реализаций.

Приведенная таблица показывает, что современная платформа Java EE опирается не менее чем на 11 дескрипторов развертывания, причем каждый из них имеет свой формат и семантику представления. В больших проектах эти файлы становятся большими и сложными для понимания, а если учесть, что они отделены от исходных текстов реализуемых компонент, то технологическая сложность проектов увеличивается во много раз.

Уникальный стиль программирования каждого разработчика не способствует упрощению технологии создания, отладки и сопровождения сервисных компонет информационных систем. Если добавить сюда проблематику сопровождения дескрипторов развертывания, то неудивительно, что первые реализации платформы Java EE пугали общественность своей сложностью.

Таблица 1.2 — Файлы дескрипторов развертывания платформы Java EE 7 [17]

Файл дескриптора	***Спецификация***	***Местоположение***
application.xml	Java EE	META-INF
application-client.xml	Java EE	META-INF
beans.xml	CDI	META-INF или WEB-INF
ra.xml	JCA	META-INF
ejb-jar.xml	EJB	META-INF или WEB-INF
faces-config.xml	JSF	WEB-INF
persistence.xml	JPA	META-INF
validation.xml	Валидация компонентов	META-INF или WEB-INF
web.xml	Сервлет	WEB-INF
web-fragment.xml	Сервлет	WEB-INF
webservices.xml	Веб-службы SOAP	META-INF или WEB-INF

Современная модель программирования компонент платформы Java EE выбирает шаблон проектирования ***Java Beans***.

> Современные парадигмы программирования предлагают переложить на контейнеры реализацию сложных и рутинных частей программного кода компонент.

В основу идеи использования шаблона Java Beans положена возможность контейнера определять структуру классов компонента за счет шаблонного именования их методов. Действительно:

а) если класс языка Java имеет ***простое свойство*** с именем **N** и типом **T**, то чтение и запись этого свойства осуществляется методами:

```
public T getN();
public void setN (T аргумент);
```

б) если тип **T** простого свойства является **boolean**, то может быть реализован метод:

```
public boolean isPropertyName();
```

в) если класс языка Java имеет ***индексированное свойство***, то можно реализовать следующие четыре метода:

```
public T getN(int индекс);
public void setN (int индекс, T значение);
public T[] getN();
public void setN (T значения[]);
```

г) если класс языка Java способен генерировать событие **T** и посылать его ***приемнику_событий***, то можно реализовать методы регистации и удаления этого приемника методами:

```
public void addTListener(TListener приемник_событий);
public void addTListener(TListener приемник_событий)
        throws java.util.TooManyListenerException;
public void removeTListener(TListener приемник_событий);
```

Более подробно об оработке событий и компоненте Java Beans можно прочитать в источнике [16, главы 24 и 37].

Повысить наглядность исходного текста компонент, а также уменьшить значимость дескрипторов развертывания или обеспечить отказ от них можно с помощью использования аннотаций.

Аннотации — это метаданные языка Java, которые наряду с дескрипторами развертывания используются для объявления и настройки сервисов.

Аннотации позволяют встраивать в исходные коды программы справочную информацию, сохраняя неизменной порядок и семантику выполнения программы, быть использована различными инструментальными средствами на стадии разработки или развертывания прикладных программ, а также обрабатываться генераторами исходного кода JAVA-программ.

Для подробного изучения назначения и области применения аннотаций рекомендуется воспользоваться материалом главы 12 источника [16].

В данной учебной дисциплине, аннотации играют важную роль, но обсуждаются по мере необходимости.

В данном пункте, мы кратко рассмотрим только общие правила их использования, определенные в пакете ***java.lang.annotation***:

а) имеется встроенная аннотация **@Retention(*правило_удержания*)**, задающая правила сохранения действия других аннотаций;

б) **правила_удержания** — правила, когда определяемая аннотация отбрасывается — перечисление ***java.lang.annotation.RetentionPolicy***, которое при-

нимает значения:

1) SOURCE — сохраняются только в исходном файле и отбрасывается при компиляции;
2) CLASS — сохраняются в файле с расширением ***.class***, но недоступны для JVM;
3) RUNTIME — сохраняются и доступны для JVM.

В качестве примера определим собственную аннотацию используемую при выполнении, с именем **MyAnno** и двумя параметрами:

а) **name** — типа ***String***;

б) **val** — типа ***int***.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
String str() default "Пример параметра";
int    val() default 100;
}
```

В дальнейшем, эта аннотация может использоваться как с новыми значениями параметров, так и без них (по умолчанию), например:

```
@MyAnno(str = "Новое значение", val = 999)
```

1.4.5 Управляемые компоненты платформы Java EE

Классическая парадигма платформы Java EE — создание контейнеров с набором служебных (технических) сервисов и предоставление разработчикам бизнес-сервисов возможность создавать компоненты, реализующие прикладную бизнес-логику.

Идейная цель классической парадигмы платформы Java EE — стимулировать разработчиков на написание слабосвязанных и максимально переносимых компонент, управляемых контейнерами. Как это делается описано в предыдущих пунктах:

а) имеется четыре вида контейнеров, в которых могут размещаться компоненты определенных типов и которые поддерживают доступ к ним по соответствующим протоколам (см. пункт 1.4.1);

б) каждый вид контейнера поддерживает соответствующий набор служебных (технических) сервисов, которыми в разной степени могут пользоваться разработчики компонент; часть этих сервисов имеют прямое при-

кладное назначение (*JSF, JPA, JAX-WS, JAX-RS и другие*), которое разработчики могут использовать как основу для реализации бизнес-сервисов (см. пункт 1.4.2);

в) каждый вид контейнера имеет свои правила формирования артефактов и соответствующие дескрипторы развертывания, обеспечивающие переносимость результатов разработки бизнес-сервисов (см. пункт 1.4.3);

г) имеется организационная структура JCP документально, с помощью публикации JSR, управляющая развитием платформы Java EE.

С позиции классической парадигмы, все компоненты платформы Java EE являются управляемыми.

Современная парадигма платформы Java EE — создание CDI-контейнеров, предоставляющих разработчикам бизнес-сервисов возможность создавать аннотируемые CDI-компоненты, реализующие прикладную бизнес-логику.

Идейная цель современной платформы Java EE — создание служебного сервиса CDI, обечечивающего контейнеры и компоненты дополнительными функциями, максимально ориентированными на использование аннотаций.

CDI (*Context and Dependency Injection*) — это «*Контекст и Внедрение Зависимостей*», представляющая реализацию ***концепции инверсии управления***, в рамках которой контейнер обеспечивает управление вашим бизнес-кодом и предоставляет технические сервисы, такие как управление транзакциями, безопасностью и другие. Для реализации сервиса CDI платформа Java EE предоставляет базовый набор JAVA-пакетов, перечисленных в таблице 1.3.

Таблица 1.3 — Базовый набор пакетов Java EE, реализующих сервис CDI [17]

Пакет	***Описание пакета***
javax.inject	Содержит базовую спецификацию по внедрению зависимостей для Java API. Описывает аннотации: @Inject, @Named, @Qualifier, @Scope, @Singleton.
javax.enterprise.inject	Основные API для внедрения зависимостей.
javax.enterprise.context	Области видимости CDI и контекстуальные API. Описывает аннотации: @ApplicationScoped, @Con-versationScoped, @Dependent, @NormalScope, @Req-uastScope, @SessionScope.
javax.enterprise.event	События CDI и API алгоритмов наблюдения. Содержит аннотацию: @Observes.
javax.enterprise.util	Пакет утилит CDI. Содержит аннотацию: @Nonbin-ding.
javax.interceptor	Содержит API перехватчика. Содержит аннотации: @AroundInvoke, @AroundTimeout, @Interceptor и другие.
javax.decorator	API декоратора CDI. Содержит аннотации: @Deco-rator и @Delegate.

Указанное программное обеспечение позволяет с помощью аннотаций сделать контейнеры платформы Java EE — CDI-контейнерами, а компоненты этой платформы — CDI-компонентами. Последней версией сервиса является CDI 2.0, а ее спецификации описаны в документе JSR 365 [19], что позволяет разработчикам компонент легко пользоваться контекстом среды исполнения и внедрять зависимости с объектами других компонент.

Сервис CDI-компонент вносит в язык Java декларативный стиль программирования в противовес императивному, требующему от программиста указания конкретных инструкций для достижения требуемого результата. Это повышает уровень абстрагирования программного обеспечения, но требует дополнительных усилий для изучения семантики используемых аннотаций.

«**Управляемые компоненты** — это объекты CDI, основанные на базовой модели управляемых компонентов. Они имеют улучшенный жизненный цикл для объектов с сохранением состояния; привязаны к четко определенным контекстам; обеспечивают сохранение безопасности типов при внедрении зависимостей, перехвате и декорации; специализируются с помощью аннотаций квалификатора; могут использоваться в языке выражений (EL)» [17].

Идейная концепция CDI имеет целью упростить использование программной платформы Java EE. Как ее использовать зависит от конкретных задач, решаемых конкретными приложениями, и будет показано в последующих главах данного учебного пособия.

1.5 Инструментальные средства реализации РСОС

Программная платформа Java Enterprise Edition — базовый framework, предназначенный для реализации масштабных сервис-ориентированных систем.

Теоретически, Java EE является одним из подходов ориентированных на создание распределенных прикладных систем (РВ-сетей, см. учебное пособие [1]), включающая в себя набор программных компонентов, позволяющих создавать полноценные сервис-ориентированные системы. В частности, среда выполнения Web-контейнеров, доступная в сети по протоколам HTTP/HTTPS и снабженная большим набором служебных (технических) сервисов, позволяет создавать стандартизированные публичные Web-службы, известные как Web-сервисы. Согласно монографии Машнина Т.С. «Web-сервисы Java» [20, стр. 7]: «Web-сервисы представляют собой программные компоненты, имеющие идентификатор URI, и взаимодействие с которыми осуществляется по Интернету с помощью открытых протоколов. Коммуникация с Web-сервисами может выполняться с помощью различных транспортных протоколов, таких как HTTP, HTTPS,

FTP, SMTP, BEEP, при этом Web-сервисы можно подразделить на три вида: SOAP Web-сервисы, ориентированные на модель RPC (вызов удаленных процедур), XML Web-сервисы, ориентированные на сообщения, и RESTful Web-сервисы».

Инфраструктура сервис-ориентированных систем предполагает наличие в сети множества доступных для потребителей сервисов серверов приложений, на которых провайдеры сервисов размещают свои приложения.

Независимо от используемой программной платформы, сервис-ориентированные системы требуют наличия:

а) аппаратных средств ВМ (ЭВМ) с установленной на ней операционной системой (ОС);

б) программных средств сервера приложений, обеспечивающих сетевой доступ к установленной на нем программной платформы содержащей ПО сервисов;

в) инструментальные средства разработки сервисов, включающие средства развертки их в среде сервис-ориентированной программной платформы.

Инфраструктура учебного процесса по данной дисциплине должна в общих чертах отражать инфраструктуру сервис-ориентированных систем.

Тематика данной дисциплины «*Распределенные сервис-ориентированные системы*» (*РСОС*) ограничена изучением технологий программной платформы Java EE, в плане последующего применения полученных знаний для целей создания Web-сервисов различного назначения. При этом предполагается, что студент имеет подготовку в пределах бакалаврского курса «*Распределенные вычислительные системы*» [1] и владеет основами программирования на языке Java в пределах платформы Standart Edition (J2SE).

Инфраструктура учебного процесса данной дисциплины ограничена учебными классами кафедры АСУ и ПО УПК АСУ на базе ОС Linux [21], дополненным:

а) индивидуальной рабочей областью студента: ***rsos-home.ext4fs.gz***;

б) СУБД Apache Derby [22];

в) сервером приложений Apache TomEE [23];

г) средой разработки Eclipse EE [24];

д) учебно-методическими пособиями (УМП) по данной дисциплине, помещенными в индивидуальную рабочую область (включая данное учебное пособие).

Эти средства обеспечивают достижение учебной цели дисциплины.

Обоснование этого утверждения изложено в следующих пяти пунктах.

1.5.1 Сервера приложений

В данной дисциплине, под сервером приложений понимается комплекс программ, реализующих программную платформу Java EE и позволяющих запускать в себе приложения Java EE.

К указанному классу относятся многие продукты, например:

а) **GlassFish** — сервер приложений с открытым исходным кодом, выпущенный компанией Sun Microsystems в июне 2005 года; в качестве контейнера сервлетов использует модифицированный Apache Tomcat;

б) **IBM WebSphere Application Server** — проприоретарный сервер компании IBM, впервые выпущенный в сентябре 1998 года; является представителем широкого спектра продукции категории middleware (промежуточного ПО), выпускаемой под брендом ***WebSphere***;

в) **JBoss Application Server** (**JBoss AS**) — сервер приложений с открытым исходным кодом, выпущенный компанией JBoss в феврале 2008 года, использующий в качестве контейнера сервлетов Apache Tomcat и преобретенный вместе с JBoss компанией RedHat в апреле 2006 года; в настоящее время выпускается под брендом **WildFly**;

г) **Apple WebObjects** — проприоретарный сервер приложений, начавший разрабатываться в марте 1996 года корпорацией NeXT Software Inc., которая в сентябре 2008 года выпустила финальную версию 5.4.3;

д) **WebLogic Server** — проприоретарный сервер приложений компании **WebLogic**, поглощенной в 1998 году компанией BEA Systems, которую в 2008 году приобрела корпорация Oracle.

В качестве учебного сервера приложений выбран дистрибутив ***Apache TomEE***, титульная страница которого приведена на рисунке 1.16.

Основными преимуществами сервера TomEE являются:

а) компактность (максимальная комплектация порядка 77 МБайт), легкое развертывание, возможность масштабирования установки, хорошая документация и совместимость со средой разработки Eclipse EE;

б) умение студентов работать с базовой частью сервера (Apache Tomcat), полученное из бакалаврского курса «*Распределенные вычислительные системы*».

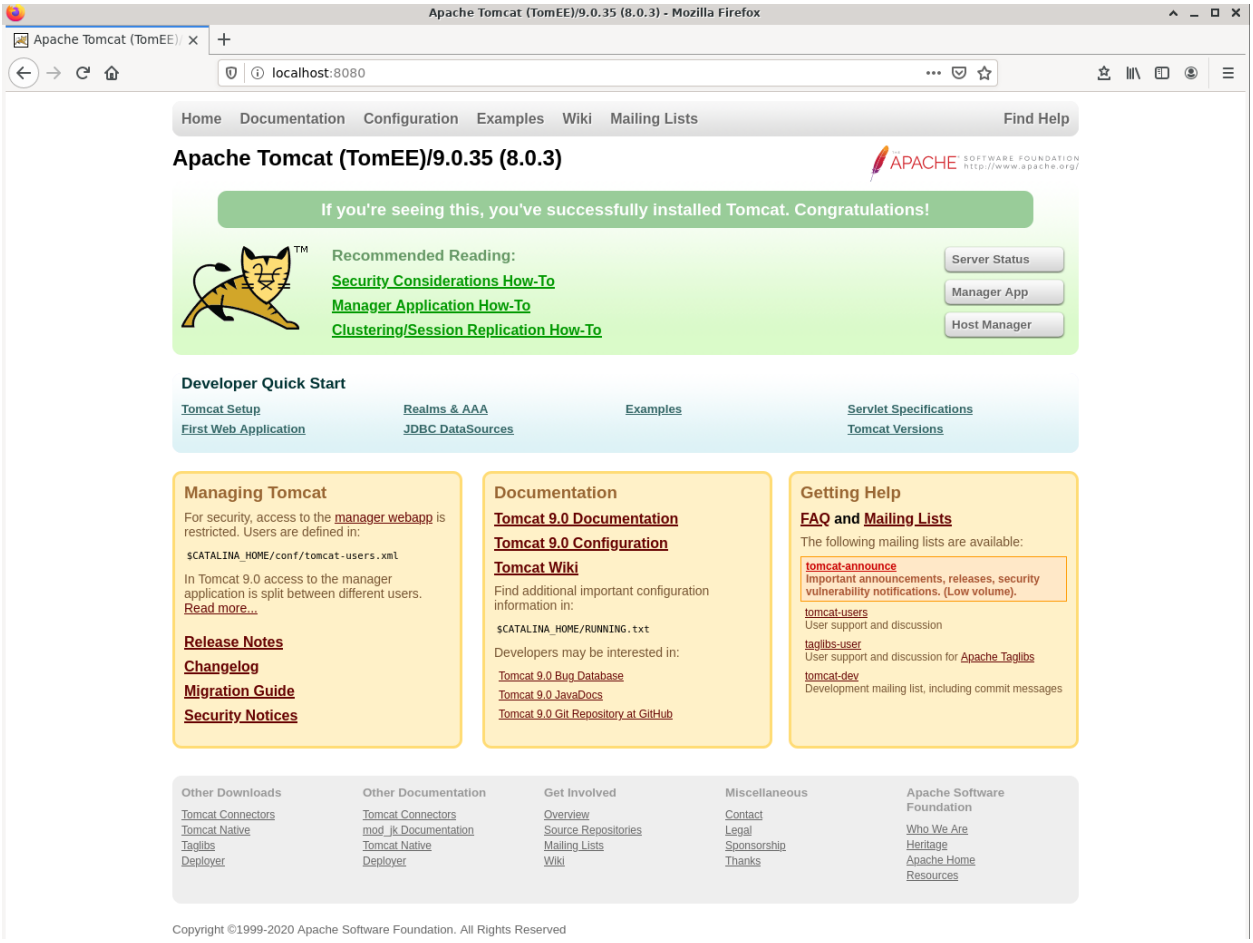

Рисунок 1.16 — Титульная страница сервера приложений Apache TomEE

Что касается недостатков, то их определить достаточно сложно, учитывая большую авторскую нестабильность приведенных выше примеров и будущую неопределенность перехода продуктов на платформу ***Jakarta EE***.

В целом, Apahe TomEE позицирует себя как Apache Tomcat + Java EE + дополнительное сервисное ПО, которое можно использовать опционально.

Таким образом, TomEE — лучший выбор для учебного процесса!

1.5.2 Микросервисы

Теоретический материал данного пособия опирается на концептуальные положения, изложенные для программной платформы Java EE 7 (июнь 2013 года) и достаточно полно представленные в монографии Э. Гонсалвеса [17]. Что касается выхода финальной платформы Java EE 8, спецификация которой была выпущена 21 сентября 2017 года, то она фактически полностью реализовала идейные положения предыдущей версии, внеся ряд несущественных для данной дисциплины изменений. Соответственно, новая платформа Jakarta EE, кроме дополнительных проблем связанных с переходом с главного домена ***javax.****

на домен ***jakarta.****, призвана продолжить развитие базовых концепций Java EE 8 с учетом новых теоретических и технологических тенденций.

Таким образом знания, полученные студентами при изучении данной дисциплины, лягут в основу будущих технологий распределенных систем.

Что касается проблематики создания больших распределенных систем, то студентам рекомендуется познакомиться с монографией С. Дашнера «*Изучаем Java EE. Современное программирование для больших предприятий*» [16], в которой, опираясь на современные достижения платформы Java EE 8, даются рекомендации по разным вопросам проектирования масштабных информационных систем.

В частности, там обсуждается современная тенденция использования ***микросервисов*** (см. [16], глава 8).

Базовая парадигма миросервисов: «… сервер приложений поставляется в контейнере, содержащем только одно приложение. Принцип «одно приложение — один сервер приложений» также соответствует идее архитектуры без разделения ресурсов» [16, стр. 302].

это утверждение обосновываются тем, что совместное использование разделяемых приложениями моделей и данных является плохой идеей, поскольку модели и данные одного приложения обычно тесно связаны друг с другом. С другой стороны, современные контейнеры серверов приложений (делается ссылка на сервер TomEE) занимают всего 150 Мбайт, что по мнению автора является значительно меньшей величиной, чем размеры самого приложения с хранилищем данных (см. [16], стр. 303).

Поскольку наша дисциплина не является курсом по проектированию распределенных систем, то вопросы подобные важности или крайней необходимости применения микросервисов далее рассматриваться не будут.

1.5.3 Apache Maven — сетевая сборка приложений

Apache Maven — официальный фреймворк для автоматизации сборки проектов платформы Java EE.

Современная парадигма развития платформы Java EE предполагает, что разработанные компоненты приложений и сервисов находятся на серверах сети и имеют свою версию, поэтому средства сборки проектов и систем сами являются распределенными системами, обеспечивающими контроль зависимо-

стей и версий.

Профессиональная разработка компонент для платформы Java EE предполагает использование ***Apache Maven***, требующего описание всех артефактов и структуры проекта в файле ***pom.xml*** (*POM, Project Object Model*). Официальная документация и описание всех примеров компонент и законченных проектов приводится в предположении, что они собраны с помощью Maven.

К сожалению инфраструктура учебного процесса кафедры АСУ и учебного программного комплекса [21] не позволяет использовать Maven по причине недоступности глобального сетевого ресурса.

1.5.4 Eclipse Enterprise Edition

Основным инструментом разработки проектов данной дисциплины является ***Eclipse Java EE IDE for Web Developers***.

Титульная заставка среды разработки Eclipse EE показана на рисунке 1.17.

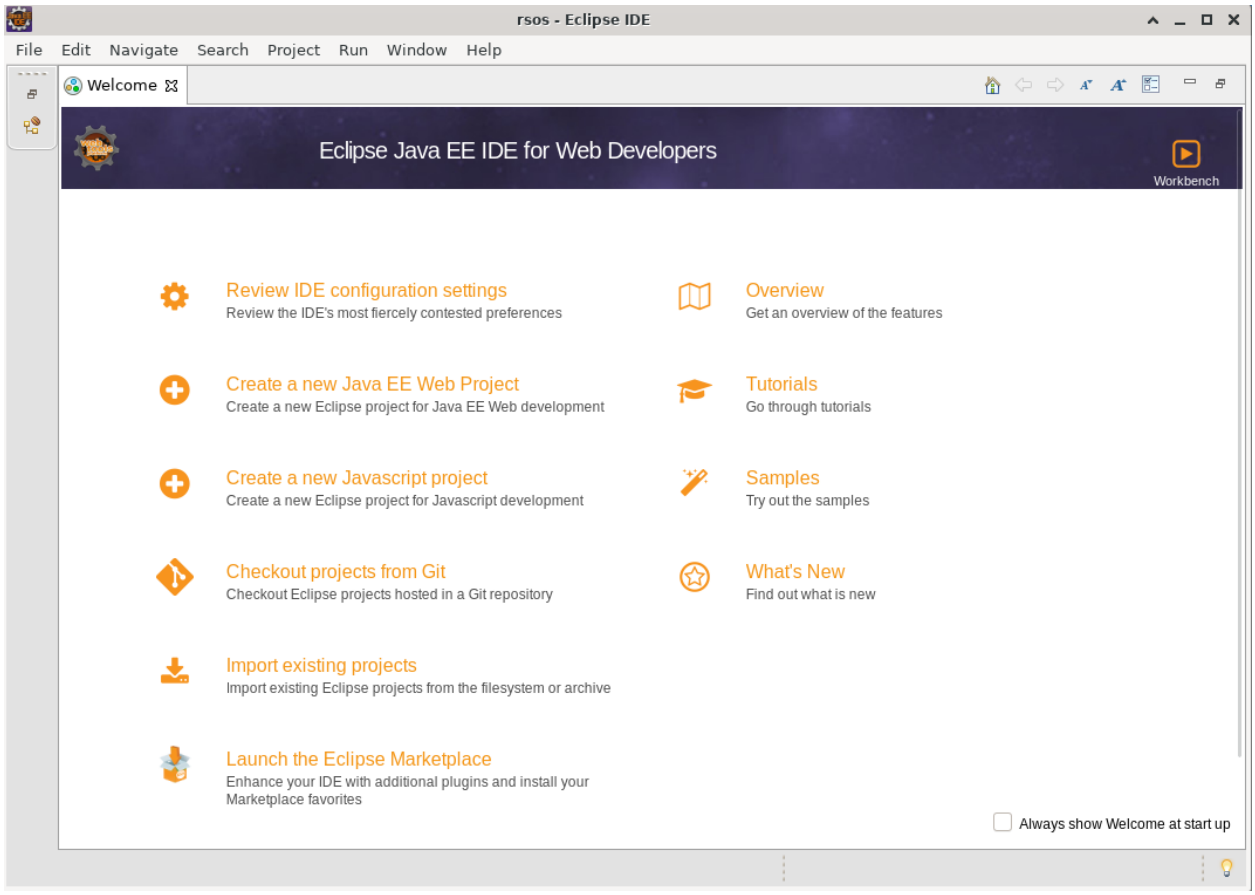

Рисунок 1.17 — Титульная заставка Eclipse EE

Среда разработки ***Eclipse EE*** хорошо знакома студентам по бакалаврскому курсу «*Распределенные вычислительные системы*», обеспечивает тесное взаимодействие своей среды с сервером приложений ***Apache TomEE***, имеет плагины для сборщиков проектов ***Ant*** и ***Maven***.

Учитывая передачу платформы Java EE под управление организации Eclipse Foundation, следует ожидать, что будущие версии данной IDE также хорошо будут поддерживать платформу Jakarta EE.

1.5.5 Тестовый пример

Все технологии, изучаемые в данной дисциплине, демонстрируются конкретными примерами.

В силу современного тренда развития сервис-ориентированных технологий и ограниченных возможностей обучающего процесса, целевая тематика изучаемой дисциплины ориентирована на Web-сервисы, которые сами являются частью Web-технологий. Программная платформа Java EE предоставляет нам все возможности для реализации таких распределенных web-сервис-ориентированных систем. Тем не менее, для демонстрации таких решений необходимо более детально изучить ряд вопросов, касающихся самих Web-технологий.

Одним из таких вопросов является возможность использования технологи HTTP-сервлетов и JSP-страниц. А поскольку студенты уже в общих чертах знакомы с ней, то сразу рассмотрим пример, который будет решать две задачи:

а) тестовую проверку правильной установки и работоспособности инструментальных средств Eclipse EE и сервера TomEE;

б) реализацию прототипа простейшего приложения, которое будет модифицироваться и реализовываться с применением других сервисных компонент платформы Java EE.

Прикладная часть приложения описывается тремя пунктами:

а) при соединении с сервером пользователю отображается форма, предлагающая ввести произвольный текст;
б) после нажатия кнопки «Отправить», в ответ выводятся: отправленное сообщение и предложение ввести новый текст;
в) в обоих случаях на экран дополнительно выводятся: тип метода сервлета, принявшего запрос, и идентификатор сессии соединения с сервером.

Реализация тестового примера обеспечивается последовательностью следующих трех действий:

а) открываем в Eclipse EE проект типа «***Dynamic Web Project***» с именем ***test***;

б) создаем в проекте сервлет с именем ***TestTomee*** и двумя методами ***doGet*** и ***doPost***, показанными на листинге 1.1;

в) создаем в проекте JSP-страницу с именем ***test.jsp*** и содержимым, показанным на листинге 1.2.

Листинг 1.1 — Исходный текст сервлета TestTomee.java проекта test

```
package asu.rsos;

import java.io.IOException;
import java.util.Date;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class TestTomee
 */
@WebServlet("/TestTomee")
public class TestTomee extends HttpServlet
{
        private static final long serialVersionUID = 1L;

        // Разделяемый ресурс
        private String msgs = "";

        /**
         * @see HttpServlet#HttpServlet()
         */
        public TestTomee() {
            super();
        }

        /**
         * @see HttpServlet#doGet(HttpServletRequest request,
         * HttpServletResponse response)
         */
        protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                              throws ServletException, IOException
        {
        /**
         * Явная установка кодировок объектов запроса и ответа.
         * Стандартная установка контекста ответа.
         */

                  response.setCharacterEncoding("UTF-8");
                  response.setContentType("text/html");
                  /**
```

```
                 * Стандартное подключение ресурса сервлета.
                 */
                RequestDispatcher disp =
                      request.getRequestDispatcher("/WEB-INF/test.jsp");
                disp.forward(request, response);
        }

        /**
         * @see HttpServlet#doPost(HttpServletRequest request,
         * HttpServletResponse response)
         */
        protected void doPost(HttpServletRequest request,
                              HttpServletResponse response)
                              throws ServletException, IOException
        {
                request.setCharacterEncoding("UTF-8");
                // Сохраняем сообщение
                Date dt = new Date();
                msgs += dt.toString() + "<br>"
                            + request.getParameter("text") + "<hr>";
                            request.setAttribute("msgs", msgs);

                // Переходим к методу doGet()
                doGet(request, response);
            }
}
```

Листинг 1.2 — Исходный текст JSP-страницы test.jsp проекта test

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
                <meta charset="UTF-8">
                <title>Тест сервера tomee</title>
      </head>
      <body>
            <hr>
                <b>Вызван метод: <%= request.getMethod() %>;</b>
                ID сессии:       <%= session.getId() %>
            <hr>
      <%
            if (request.getMethod().equals("POST")){
                      out.println(request.getAttribute("msgs"));
            }
      %>

            <form action="TestTomee" method="post" accept-charset="UTF-8">
                <p> Введи текст: <br>
                      <textarea rows="5" cols="40" name="text"></textarea>
                </p>
                <p>
                      <input type="submit">
                </p>
            </form>
      <hr>
```

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

Анализ полученного результата проведем, демонстрируя запуск созданного приложения.

Результат первого запуска показан на рисунке 1.18, из которого видно, что:

a) сервер обработал запрос по методу GET;
b) вызов имеет собственный идентификатор сессии.

Если последовательно отправить на сервер два сообщения «*Сообщение 1*» и «*Сообщение 2*», то получим результат, показанный на рисунке 1.19, из которого следует, что:

а) тестовое приложение работает согласно поставленной задаче;

б) все запросы сервер обрабатывает в пределах одной сессии;

в) сервлет ***TestTomee*** кэшируется в памяти сервера, накапливая все полученные сообщения.

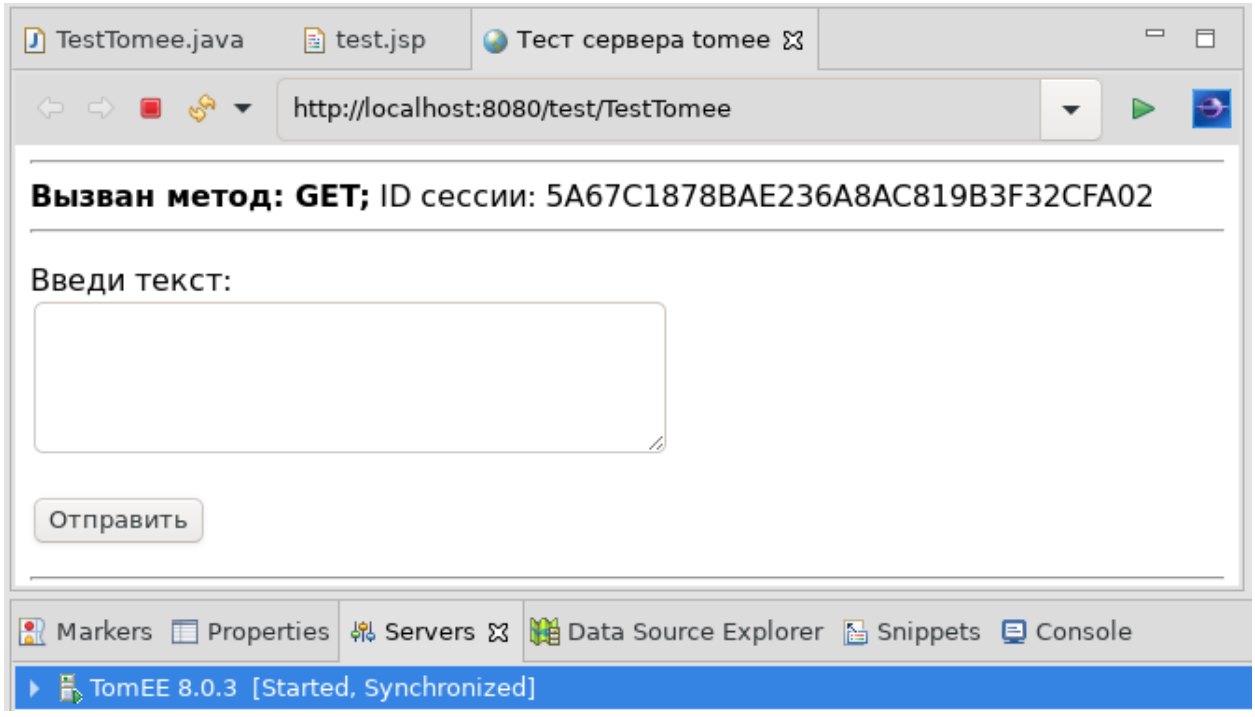

Рисунок 1.18 — Первый запуск тестового примера

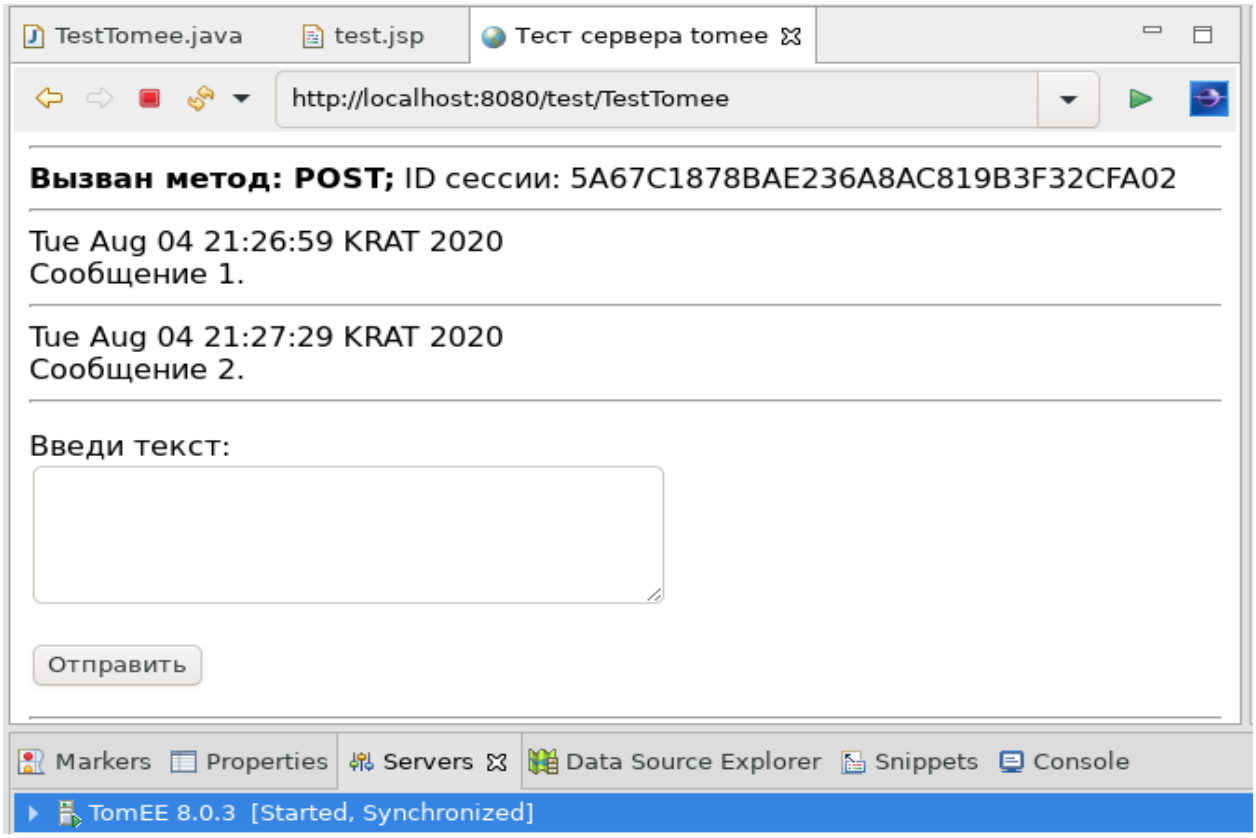

Рисунок 1.19 — Результат отправки на сервер двух сообщений

Теперь перейдем к выводам по результатам первой главы.

1.6 Заключение по первой главе

Дисциплина «*Распределенные сервис-ориентированные системы*» содержит знания накопленные социумом в результате диалектического развития двух альтернативно направленных процессов:

– ***нисходящего процесса проектирования***, отражающего общее стремление иметь и использовать целевые масштабные информационные системы;
– ***восходящего процесса реализации***, отражающего общее стремление создавать эффективные и управляемые системы.

Изучение любой дисциплины, проходящей в рамках собственного учебного процесса, формируются свои проекции (видения) на указанные выше процессы проектирования и реализации систем.

Традиционно считается, что процессы проектирования яляются концептуально более главными чем процессы реализации, поскольку они отражают целевые установки по достижению нужного результата. Такой же традиции придерживается и структура учебного материала данного пособия, где первая глава

отражает теоретические построения связанные с процессами проектирования масштабных информационных систем, а последующие главы раскрывают отдельные процессы реализации таких систем.

1.6.1 Итоги теоретических построений первой главы

Основные теоретические построения распределенных сервис-ориентированных систем (РСОС) сосредоточены в первых трех подразделах.

Подраздел 1.1 рассматривает этапы развития распределенных систем, которые в классическом варианте ограничены объектным подходом и достаточно подробно изучены в бакалаврском курсе «*Распределенные вычислительные системы*». Важность этой части учебного материала определяет явное осознание проблем, которые были вызваны восходящими процессами реализации объектных систем.

Подраздел 1.2 рассматривает период становления РСОС, который связан с развитием процессов реализации Web-систем, занимающих в свое время альтернативную позицию по отношению к объектному подходу. Особая важность этой части учебного материала — классическое представление об архитектуре SOA, появление протокола SOAP и реализация РСОС первого поколения.

Подраздел 1.3 рассматривает кризисную часть концепции РСОС, которая связана с построением универсальной модели SOA. Здесь изложены четыре точки зрения: эталонная модель SOA, модель Захмана, модель СОС POSIX и бизнес-парадигма модели SOA, отражающая современную тенденцию развития РСОС.

Основные теоретические построения прикладного аспекта реализации РСОС сосредоточены в следующих двух подразделах.

Подраздел 1.4, посвященный программной платформе Java EE, раскрывает общий теоретический подход контейнерной архитектуры распределенных систем, обеспечивающий достаточно приемлемую реализацию сложных широко масштабируемых информационных систем. В частности, на этой платформе проведена реализация многих Web-сервисных систем. В силу отсутствия значимых альтернатив, эта платформа выбрана в качестве концептуальной и технологической основы изучаемой дисциплины.

Подраздел 1.5 описывает инфраструктурную часть учебного процесса изучаемой дисциплины, которая посредством инструментальных средств сервера приложений Apache TomEE, IDE Eclipse EE и СУБД Apache Derby, обеспечивает разработку и исследование приложений РСОС.

1.6.2 Тематический план последующих глав

Теоретический материал первой главы дает только общее концептуальное представление о предметной области изучаемой дисциплины. Его еще недостаточно для построения даже простейших Web-сервисов, поэтому последующие три главы посвящены изучению различных аспектов Web-технологий, обеспечиваемых программной платформой Java EE.

Глава 2 концентрирует внимание на компоненте JSF (JavaServer Faces). На ее основе успешно реализуется не только уровень бизнес-логики сервисных систем, но и уровень интерфейса сервисов (см. пункт 1.3.4, рисунок 1.12). Сама реализация компоненты JSF основана на последних технологических достижениях платформы Java EE, что делает ее хорошим учебным примером, раскрывающим на нужном уровне концепции создания и использования идеи самих программных компонент, управляемых контенерами сервера.

Глава 3 дополняет учебный матерал главы 2 в плане изучения современных технологий хранения и использования данных. В ней рассматриваются вопросы программирования взаимодействия с реляционными СУБД, отличные от использования языка SQL. В этом плане программная платформа Java EE содержит инструменты работы с объектными сущностями посредством сервиса Java Persistence API (*JPA*), поддерживающего «механизмы» объектно-реляционного отображения (*ORM, Object-Relational Mapping*).

Глава 4 завершает описание различных технологий платформы Java EE, необходимых для реализации Web-сервисов. Прежде всего это касается работы с представлениями в формате XML, составляющих основу протокола SOAP. Дополнительно рассматриваются популярные в настоящее время спецификации представления JSON.

Основной учебный материал данной дисциплины, посвященный системам РСОС, изложен в последних двух главах.

Глава 5 посвящена классическому представлению Web-сервисных систем, основанных на протоколе SOAP. Здесь рассмотрены интеграционные процессы объединения различных сервисов, широко известные как business-to-business (*B2B*) и Enterprise Application Integration (*EAI*). Подробно обсуждаются элементы такой интеграции, основанные на спецификациях XML, WSDL, SOAP и UDDI.

Глава 6 представляет учебный материал, демонстрирующий технологию Web-служб в стиле REST. Данная технология считается современной, перспективной и лишенной многих недостатков, которые присущих классическому подходу на базе протокола SOAP.

Учебные примеры, приведенные в данной дисциплине, не претендуют на эталонные реализации компонент РСОС.

Все учебные примеры данного методического пособия, включая тестовый пример предыдущего подраздела, имеют своей целью демонстрацию технологических возможностей программной платформы Java EE или демострацию иных возможностей распределенных сервис-ориентированных систем. Даже, если в тексте пособия приводится некоторая прикладная интерпретация решаемой задачи, ее не следует рассматривать как эталонное решение, требующее подражания. Более того, прикладная часть приведенных примеров сознательно упрощена для лучшего понимания технологических аспектов рассматриваемых вопросов.

Несмотря на высокую переносимость программного обеспечения разработанного на языке Java, следует максимально внимательно относиться к версиям используемых компонент.

Данное замечание следует рассматривать как формальное предупреждение, поскольку современные тенденции разработки ПО ориентированы на использование инструментальных средств поддерживающих версионность используемых библиотек и иных внешних программных продуктов.

Вопросы для самопроверки

1. На какие две группы делит системы классификация СОД?
2. Какое программное обеспечение обязательно присутствует в обобщенной модели РВ-сетей?
3. Какое программное обеспечение обязательно содержит классическая модель взаимодействия «Клиент-сервер»?
4. Какая архитектура соответствует объектному подходу в распределенных системах?
5. Что такое — связность распределенных сервис-ориентированных системам?
6. Какова концептуальная суть архитектуры SOA?
7. Каковы базовые концепции эталонной модели SOA?
8. В чем состоит основная идея концепции Захмана?
9. Что из себя представляет эталонная модель СОС POSIX?
10. Сколько и какие уровни предлагает бизнес-парадигма модели SOA?
11. Что из себя представляют контейнеры и компоненты программной платформы Java EE?
12. В чем состоит отличие понятия EJB от понятия Java Beans?
13. Что такое — артефакты контейнеров?
14. Что такое — аннотации и чем они отличаются от дескрипторов развертывания?
15. Что такое — управляемые компоненты платформы Java EE?
16. Что такое — CDI и в чем состоит их назначение?
17. Что такое — сервер приложений и для каких целей он используется?
18. Для каких целей используются микросервисы?
19. Для каких целей используется Apache Maven?
20. Что такое — Jakarta EE и в чем ее отличие от Java EE?

2 Тема 2. Использование компоненты JSF контейнера Web

Документальное представление информации — сервис, потребителями которого являются бизнес-руководители предприятий, а поставщиками — IT-специалисты различных организаций.

Следуя бизнес-парадигме архитектуры предприятия, уже описанной ранее в пункте 1.3.4 главы 1 (см. рисунок 1.12), мы полагаем, что бизнес-руководители предприятий реализуют уровень бизнес-логики, планируя свои решения и осуществляя свою деятельность на основе документального представления поступающей к ним информации. Если для отображаемой бизнес-информации необходимо сложное и специальное графическое представление, то для этой цели необходимо разработать специальное приложение, например, некоторый АРМ-руководителя. Если для представления бизнес-информации достаточно шаблонного представления в виде набора управляемых форм документов, то для этих целей достаточно приложения браузер.

В последнем случае, идеальным решением для поставщика сервиса может оказаться использование сервера приложений, на основе программной платформы Java EE, а в качестве основы реализации такого сервиса отображения бизнес-информации может стать компонента платформы — JSF.

Учебная цель данной главы — изучение технологии создания управляемых компонент CDI, на примере использования служебного (технического) сервиса JSF-компоненты Web-контейнера сервера приложений.

Как было отмечено в подразделе 1.4 предыдущей главы, для многих разработчиков программного обеспечения платформа Java EE кажется слишком сложной, поэтому они отказываются от нее в пользу альтернативных решений. Использование управляемых компонент (*CDI, Context and Dependency Injection*) устраняет указанный выше недостаток, делая платформу Java EE вполне конкурентноспособным инструментом реализации распределенных сервис-ориентированных систем (*PCOC*).

В контексте тематики данной главы, CDI рассматривается под углом применения ее как к компоненте JSF, так и к учебным примерам, использующим эту компоненту. В любом случае, студент должен внимательно следить за дуализмом толкования семантики понятия «*Контекста и внедрения зависимостей*»:

а) как служебного сервиса CDI для контейнера платформы Java EE;

б) как объекта CDI, реализующего функциональность приложения.

2.1 Web-сервис представления бизнес-информации

В плане использования Web-технологий JSF конкурирует с технологией HTTP-сервлетов и JSP-страниц.

Официально, для представления информации программная платформа Java EE использует технологию JSF (*JavaServer Faces*). В плане тематики данной главы и архитектуре сетевого взаимодействия «*Клиент-сервер*», такое представление информации соответствует системной архитектуре «*Тонкий клиент*», когда на стороне потребителя информации используется только приложение браузера.

Общая терминология и проблематика начального этапа развития Web-технологий были описаны в подразделе 1.2 первой главы, а здесь мы рассмотрим детали реализации системной архитектуры «*Тонкий клиент*», применительно к тематике РСОС.

2.1.1 Языки HTML, JavaScript и протокол HTTP

Традиционно браузер, как приложение клиента, отображает информацию в формате HTML, которую он получает с Web-сервера. На рисунке 2.1 показан файл ***web.xml*** — дескриптор развертывания проекта ***test***.

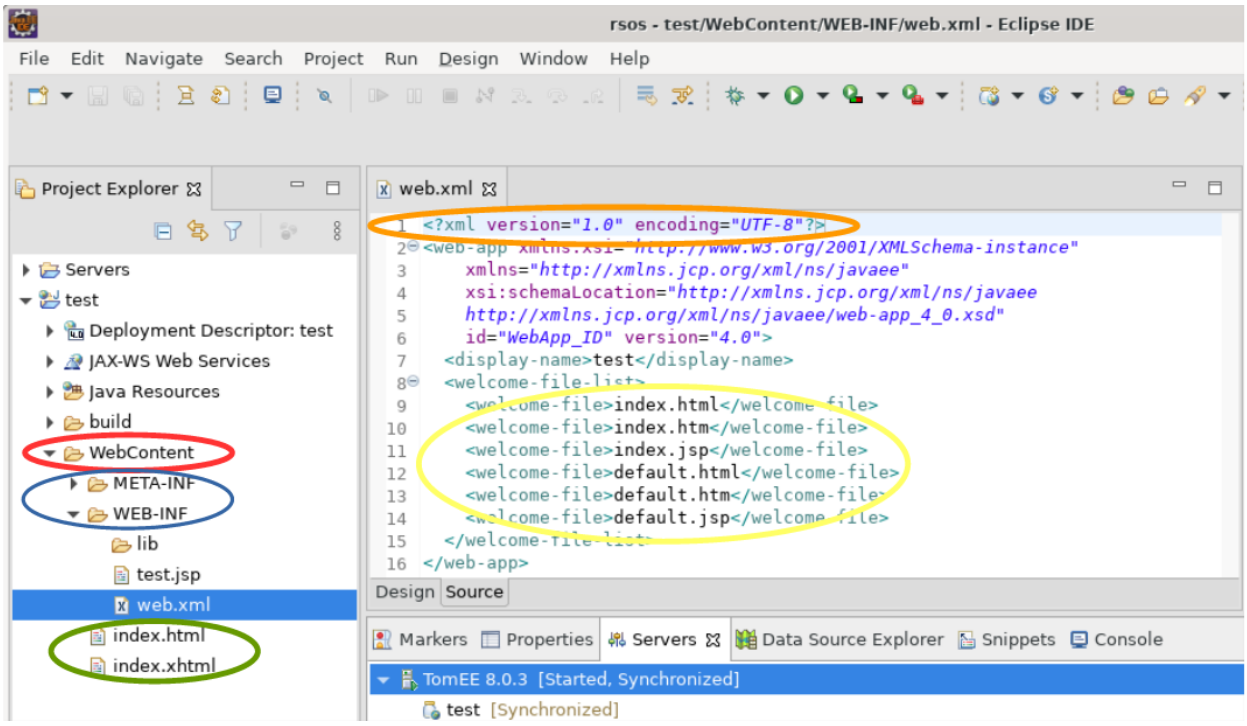

Рисунок 2.1 — Дескриптор тестового приложения — web.xml

Файл ***web.xml*** имеет формат XML-файла, на что указывает его верхняя строка выделенная оранжевым цветом. Желтым цветом выделены имена файлов, которые могут запускаться по умолчанию (в порядке их перечисления), если пользователь обратился по адресу: ***http://localhost:8080/test/***.

Web-сервер обеспечивает свободный доступ ко всем файлам, помещенным в каталог ***WebContent*** (выделено овалом красного цвета) или его подкаталоги, кроме служебных подкаталогов ***META-INF*** и ***WEB-INF*** (выделено овалом синего цвета). Для примера, овалом зеленого цвета выделены доступные файлы проекта ***test***, причем файл ***index.html***, показанный на рисунке 2.2, будет запускаться по умолчанию.

web.xml | index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>Дисциплина РСОС</title>
6  </head>
7  <body>
8      <h2>Распределенные сервис-ориентированные системы</h2>
9      <hr>
10     <h3>Приветствуем Вас на сайте дисциплины РСОС!</h3>
11     <br>
12     Для запуска тестового примера перейдите по ссылке:
13     <a href="http://localhost:8080/test/TestTomee">
14     Сервлет TestTomee</a>
15 </body>
16 </html>
```

Рисунок 2.2 — Содержимое файла index.html

Выделенная на рисунке строка <!DOCTYPE html> указывает, что используется язык HTML версии 5, а содержимое файла предлагает запустить по ссылке тестовый сервлет, как это показано на рисунке 2.3.

Современные браузеры прекрасно отображают простейшие файлы формата XHTML.

На рисунке 2.4, в файле ***index.xhtml***, представлено тоже самое прикладное содержимое, но в формате XHTML. Это видно из выделенного заголовка, где строка "***http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd***" требует, чтобы все теги разметки обязательно имели соответствующие закрывающие теги, но позволяется использовать презентационные элементы, такие как ***center***, ***font*** и ***strike***, которые недопустимы в строгой версии.

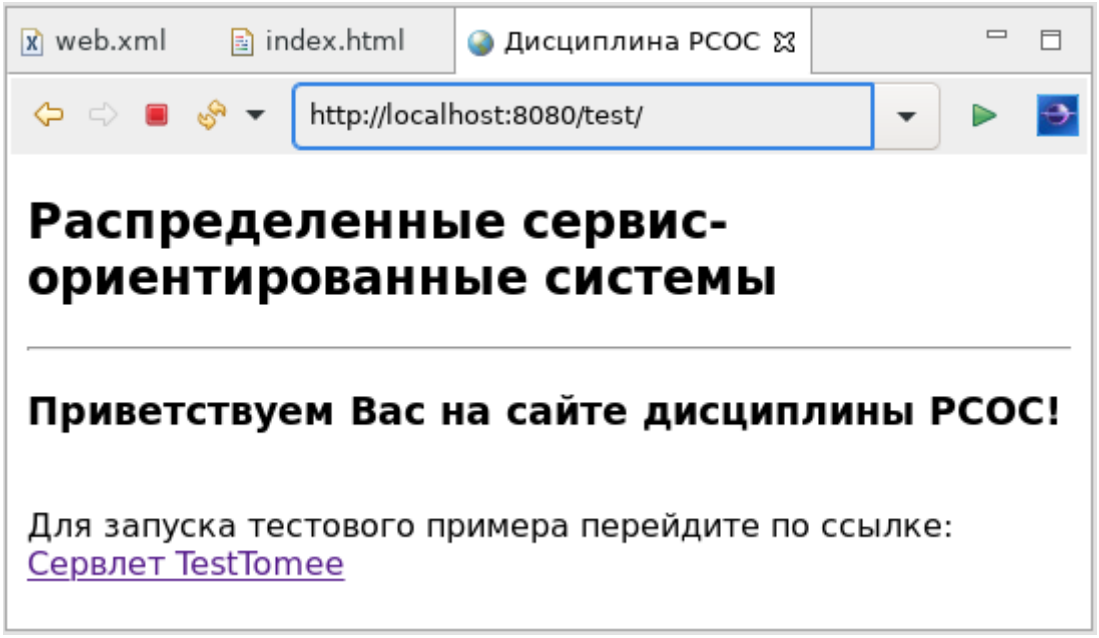

Рисунок 2.3 — Результат отображения файла index.html

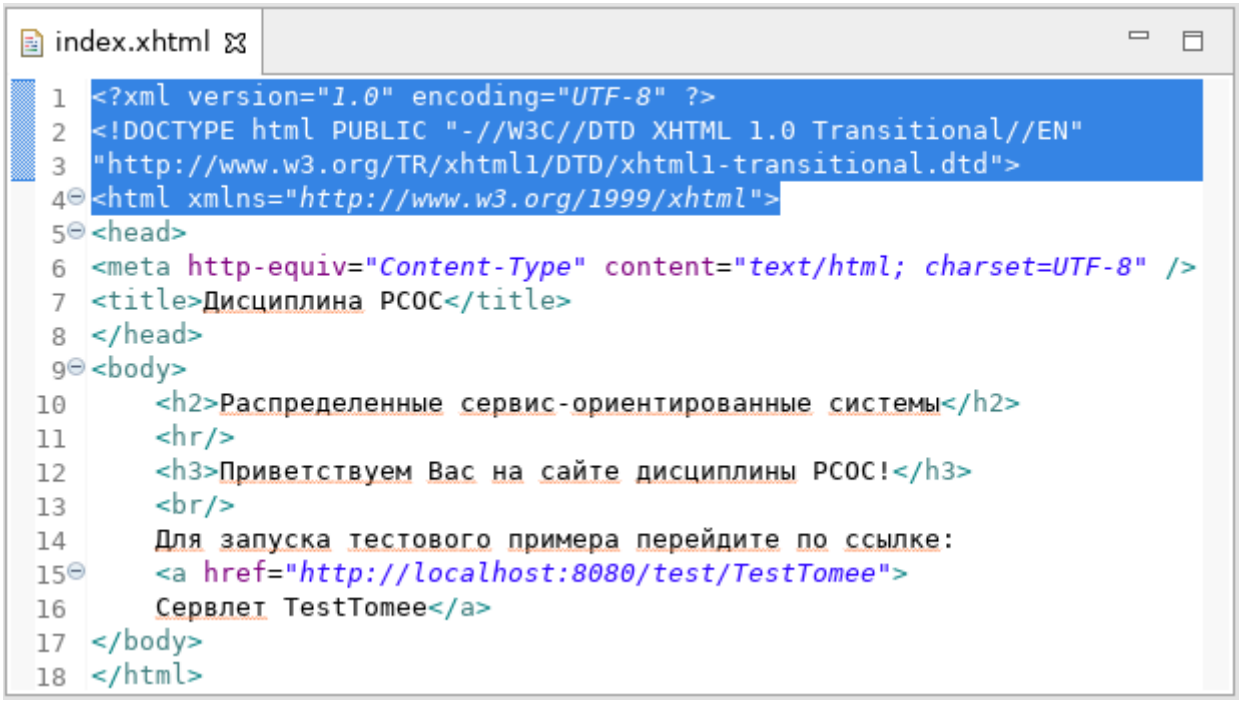

Рисунок 2.4 — Содержимое файла index.xhtml

В общем случае, могут быть использованы и другие форматы проверки файлов формата XHTML:

а) "***http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd***" — переходный вариант, который также позволяет определzть набор документов;

б) "***http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd***" — наиболее ограничительный вариант, строго следующий спецификации HTML 4.01.

С прикладной точки зрения оба файла одинаковы и, при отображении информации файла ***index.xhtml***, будет показано тоже, что и на рисунке 2.2. Но, если мы посмотрим содержимое страницы (см. рисунок 2.5), то оно будет несколько другим.

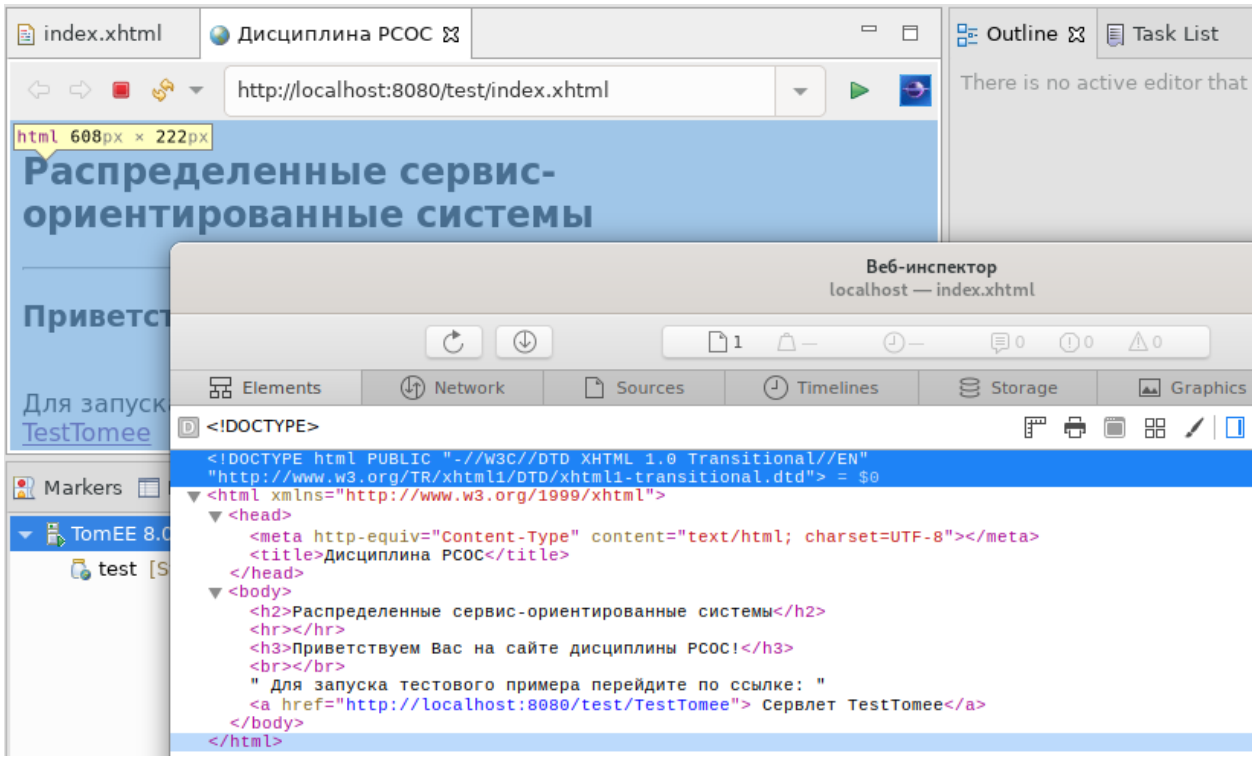

Рисунок 2.5 — Отображаемое содержимое файла index.xhtml

Что касается языка ***JavaScript*** (не путать с технологией ***AJAX***), то его назначение — работа в среде виртуальной машины браузера. Хотя эти возможности кажутся привлекательными, основная проблема здесь связана с безопастностью проектных решений. В свое время JavaScript вытеснил технологию JAVA-апплетов и получил достаточно большую популярность, благодаря своей широкой доступности. Тем не менее, у него имеются три проблемы:

а) прямое его использование невозможно по причине публикации им всех средств доступа к используемым ресурсам;

б) прогрессивно увеличивается сложность отображаемых страниц, поскольку функции и данные становятся частью страниц;

в) увеличивается связность отдельных страниц и ресурсов серверных приложений, что также приводит к сложности реализации уровня интерфейсов сервисной архитектуры приложений.

Классический протокол HTTP предоставляет клиенту и серверу восемь методов взаимодействия: GET, POST, OPTIONS, HEAD, PUT, PATCH, DELETE, TRACE и CONNECT. С прикладной точки зрения наиболее популярными являются методы GET и POST, на которые и ориентируются сервис-ориентированные системы, использующие только браузеры.

2.1.2 Серверные технологии PHP и HttpServlet

Современная парадигма РСОС — серверные технологии.
Студенту настоятельно рекомендуется повторить главу 4 «*Web-технологии распределенных систем*» учебной дисциплины [1].

Современная парадигма распределенных систем ориентирована на трехзвенные архитектуры «*Клиент-сервер*», что показано на рисунке 2.6.

Рисунок 2.6 — Трехзвенная архитектура «Клиент-сервер» [1]

Для технологий PHP и HttpServlet, эта трехзвенная архитектура интерпретируется рисунком 2.7.

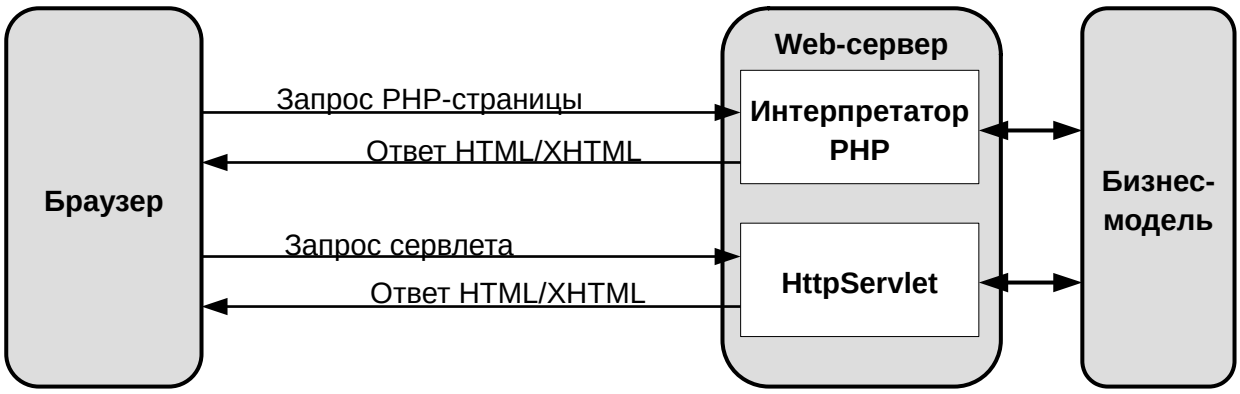

Рисунок 2.7 — Интерпретация трехзвенной архитектуры

Трехзвенная архитектура «*Клиент-сервер*» освобождает приложение клиента от необходимости детального программирования запросов с помощью языка JavaScript. Это не только повышает безопастность использования web-технологий, но также упрощает HTML/XHTML-ответ принимаемый браузером.

Практическое использование компоненты HttpServlet имеет явное преимущество над PHP-интерпретатором, которое выражается двумя пунктами:

а) PHP-интерпретатор проводит ***синтаксический анализ*** запрашиваемой PHP-страницы при каждом запросе браузера;

б) HttpServlet является исполняемым классом языка Java, поэтому он уже прошел синтаксический контроль и ***кэшируется Web-конейнером*** сервера приложений.

Технология сервлетов реализует шаблон проектирования MVC.

Технология сервлетов, фактически напрямую, реализует шаблон проектирования «*Model-View-Controller*» (*MVC*), что наглядно показано на рисунке 2.8. Более подробно (см. [1, глава 4, пункт 4.3.5, «*Модель MVC*»]).

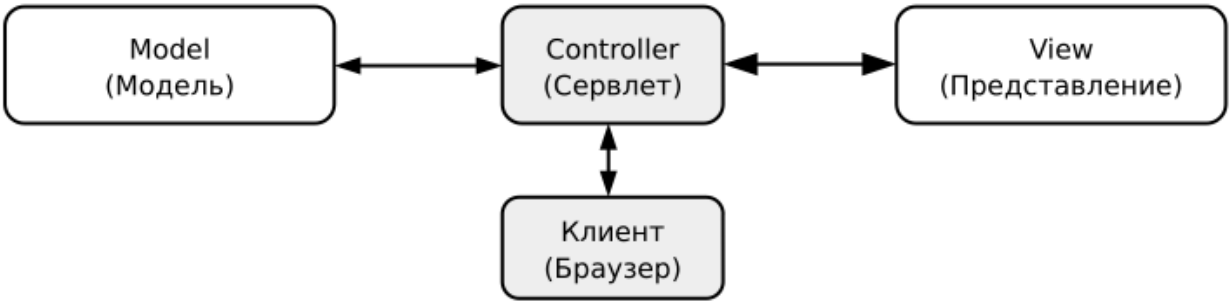

Рисунок 2.8 — Трехзвенная архитектура, реализуемая моделью MVC [1]

Для наглядности, покажем трехзвенную архитектуру и шаблон MVC, реализованный ранее, в пункте 1.5.5 для тестового примера проекта ***test***.

Сам контроллер реализован сервлетом ***TestTomee*** (см. листинг 1.1), функциональность которого обспечивается стандартными методами ***doGet(...)*** и ***doPost(...)***. **Модель шаблона** реализована внутри сервлета в виде приватного разделяемого ресурса (строковый объект ***msgs***), как это показано ниже.

```
/**
 * Servlet implementation class TestTomee
 */
@WebServlet("/TestTomee")
public class TestTomee extends HttpServlet {
	private static final long serialVersionUID = 1L;

	// Разделяемый ресурс
	private String msgs = "";
```

Представление (View) выполняет JSP-страница ***test.jsp***, ранее приведенная на листинге 1.2 и содержащая показанные ниже JSP-конструкции.

```
</head>
        <body>
        <hr>
                <b>Вызван метод: <%= request.getMethod() %>;</b>
                ID сессии:       <%= session.getId() %>
        <hr>
<%
if (request.getMethod().equals("POST")){
                out.println(request.getAttribute("msgs"));
}
%>
        <form action="TestTomee" method="post" accept-charset="UTF-8">
```

Если теперь запустить тестовый пример подобно тому, как это показано на рисунке 1.19, то полученная браузером страница будет иметь вид, показанный на рисунке 2.9.

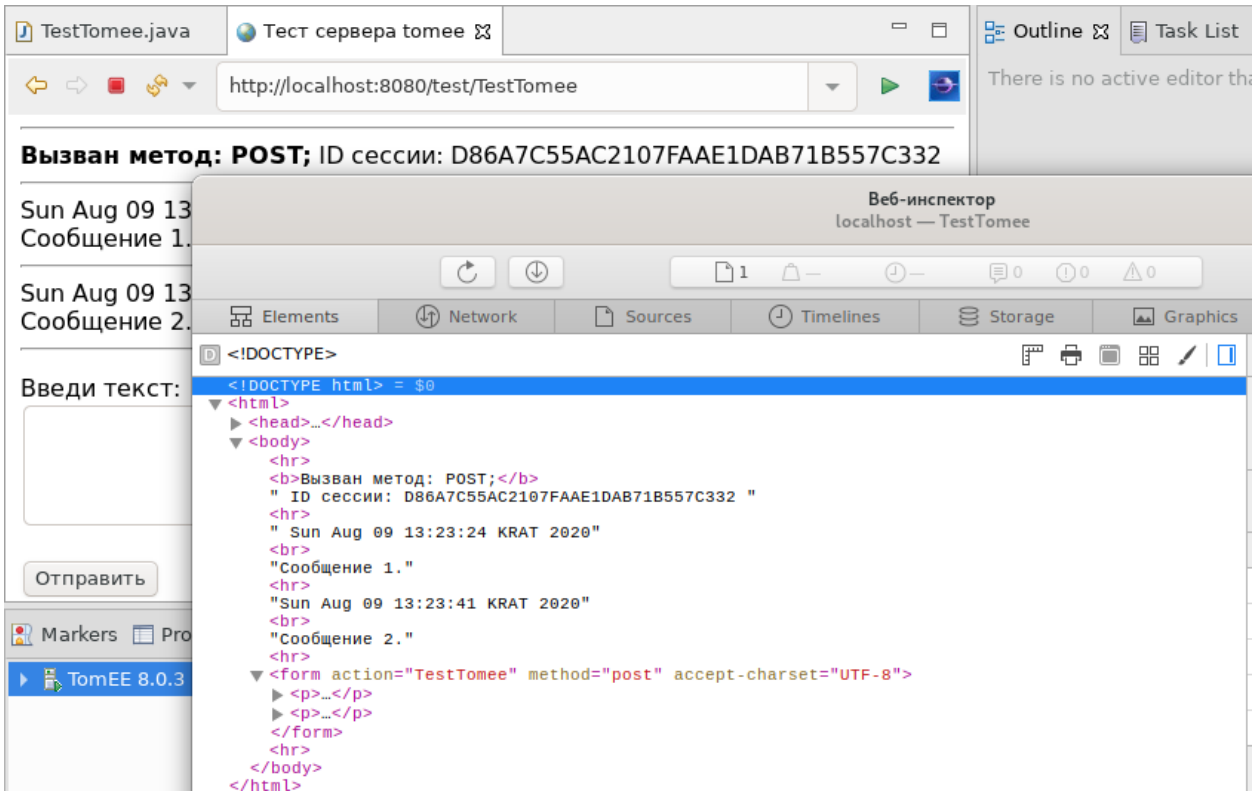

Рисунок 2.9 — HTML-страница тестового примера, полученная браузером

К общим недостаткам технологий PHP и HttpServlet следует отнести:

а) ***полная замена*** HTTP-страницы при каждом обращении к серверу;

б) ***полностью «ручная»*** реализация контроллера MVC.

2.1.3 Технология AJAX и компонента JavaServer Faces

Общая проблема классической концепции web-технологий — необходимость синхронного способа взаимодействия браузера и Web-сервера, предполагающего загрузку новой страницы при каждом HTTP-запросе браузера.

В 2005 году, Джесси Джеймс Гаррет предложил, а разработчики браузеров реализовали новую технологию AJAX.

AJAX (*Asynchronous Javascript and XML*) — асинхронный JavaScript и XML, предполагающий наличие в браузере специальной библиотеки (обработчика), доступной с HTML-страницы на языке JavaScript и обеспечивающей асинхронное взаимодействие с Web-сервером.

На рисунке 2.10 показано различие способов классического взаимодействия браузера с Web-сервером и с помощью технологии AJAX.

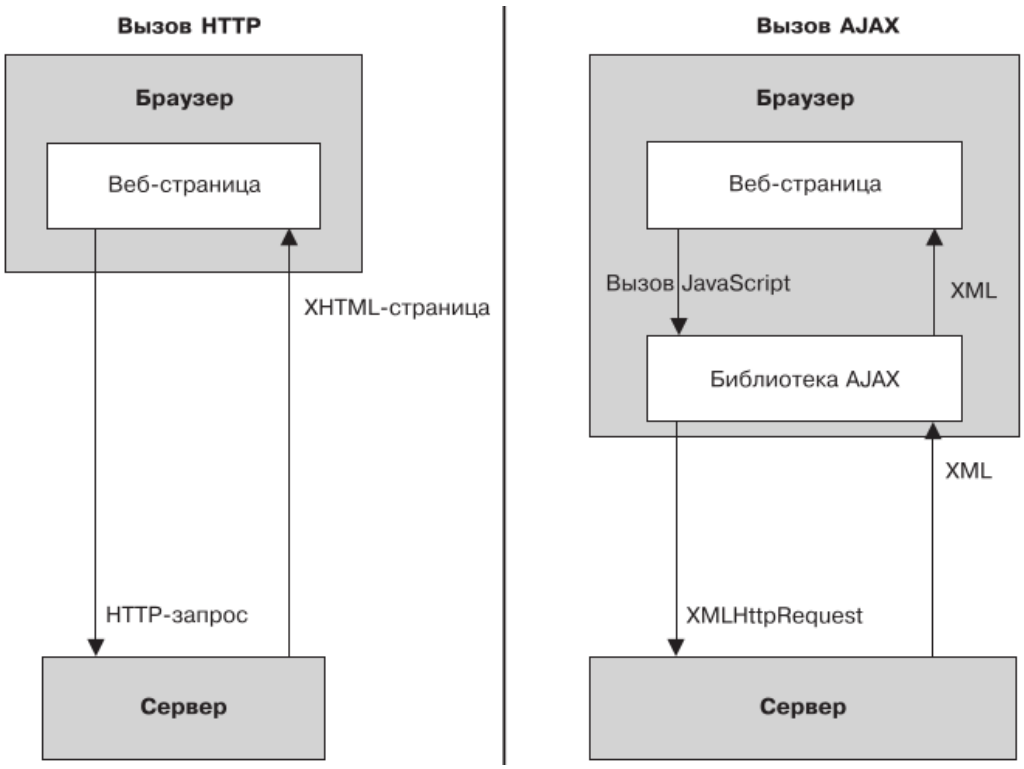

Рисунок 2.10 — Классический- и AJAX-способы взаимодействия браузера и Web-сервера

Несмотря на достаточно обширный список недостатков, которые мы рассматривать не будем, AJAX имеет ряд преимуществ: экономия сетевого трафика и сетевой нагрузки на сервер, ускорение реакции интерфейса, возможности интерактивной обработки запросов браузера и возможность использования средств мультимедиа. Неудивительно, что технология AJAX получила широкую популярность и существенно снизила интерес к технологии сервлетов.

Программная платформа Java EE выбирает технологию JSF.

В марте 2004 года, компания Sun Microsystems выпустила первую версию спецификации JSF 1.0. В октябре 2009 года, спецификация JSF была полностью пересмотрена с учетом опыта разработки и представлена как JSF 2.0. Последняя спецификация JSF 2.3 была выпущена в конце марта 2017 года. Она и является предметом нашего изучения.

Цель проекта JSF — устранение недостатков, присущих технологии JSP-страниц согласно уникальной архитектуре показанной на рисунке 2.11.

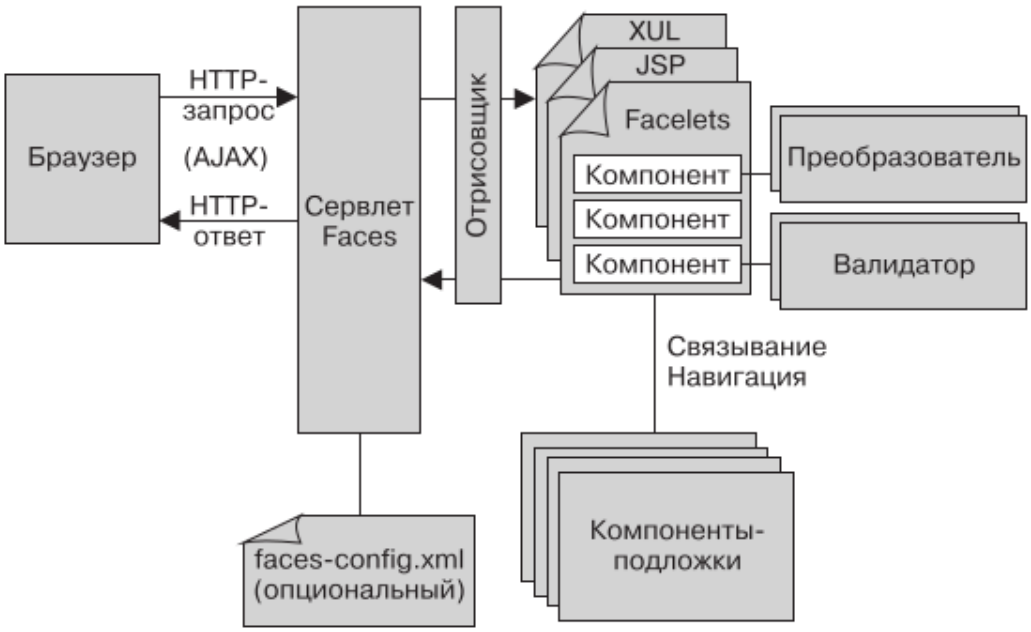

Рисунок 2.11 — Общая архитектура технологии JSF [17]

JSF является компонентой Web-контейнера платформы Java EE, где центральное место занимает специальный сервлет Faces (***FacesServlet***), который опционально может настраиваться собственным дескриптором развертывания в виде файла ***faces-config.xml***.

JSF обрабатывает все HTTP-запросы браузера, включая запросы технологии AJAX, что делает ее современной конкурентной технологией.

Компонента JSF реализует шаблон проектирования MVC, где ***контроллер*** реализуется самим сервлетом, ***представления*** — менеджером ***Facelets***, а ***модель*** — компонентами-подложками.

2.2 Шаблон проектирования MVC

Компонента JSF проектировалась по шаблону MVC.

На рисунке 2.12 компонента JSF в явном виде представлена как шаблон проектирования MVC, где сам FacesServlet выполняет роль контроллера. Если сравнить этот рисунок с рисунком 2.8 (см. стр. 67), представляющим трехзвенную архитектуру MVC, то мы увидим много общего. Более того, далее (в пункте 2.1.2) было показано, как тестовый пример можно представить в виде шаблона MVC.

Если смотреть еше раньше, то в пункте 1.3.4 главы 1 на рисунке 1.12 (стр. 32) представлена трехуровневая бизнес-парадигма предприятия. Ее также можно представить шаблоном MVC:

а) **Контроллер** — бизнес-руководители предприятия;

б) **Представление** — уровень бизнес-логики;

в) **Модель** — уровень интерфейсов сервисов.

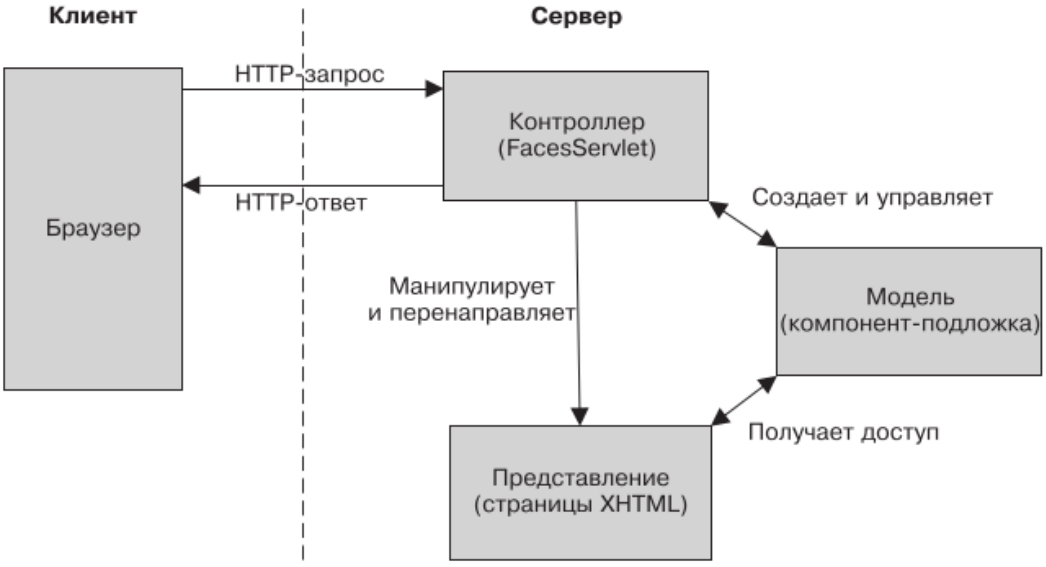

Рисунок 2.12 — JSF в виде шаблона проектирования MVC [17]

Хотя наша дисциплина не является прямым курсом по проектированию информационных систем, каждый IT-специалист должен владеть шаблоном проектирования MVC, поскольку он дает важную декомпозицию для анализа и обоснования принимаемых решений.

Несмотря на явное подобие назначения контроллеров ***HttpServlet*** и ***FacesServlet***, связанных с обработкой HTTP/XHTTP/AJAX-запросов, между ними имеются существенные различия:

а) **HttpServlet** предоставляет браузеру вызывать себя с помощью ***URI-адреса***, предоставляя программисту несколько методов (обычно ***doGet(...)*** и ***doPost(…)***); программист сам решает, когда он обращается к модели, когда — к представлению, а когда формирует ответ браузеру;

б) **FacesServlet** управляется Web-контейнером и позволяет браузеру с помощью ***URI-адреса*** обратиться к некоторому XHTML-ресурсу, который посредством шаблонов ***Facelets*** (см. рисунок 2.11) вызывет скрытые от браузера XHTML-ресурсы (***компоненты***); каждая из ***Facelets-компонент*** может представлять статический XHTML-ресурс или быть связанной с одной или несколькими компонентами-подложками.

Учебная цель данного подраздела — более подробное раскрытие общей архитектуры технологии JSF, представленной рисунком 2.11, используя в качестве логической базы шаблон проектирования MVC (рисунок 2.12).

2.2.1 Контроллер FacesServlet и жизненный цикл запроса

Каждый запрос браузера обрабазывается отдельным процессом (потоком) виртуальной машины Java.

Рисунок 2.13 показывает структурную схему обработки запроса.

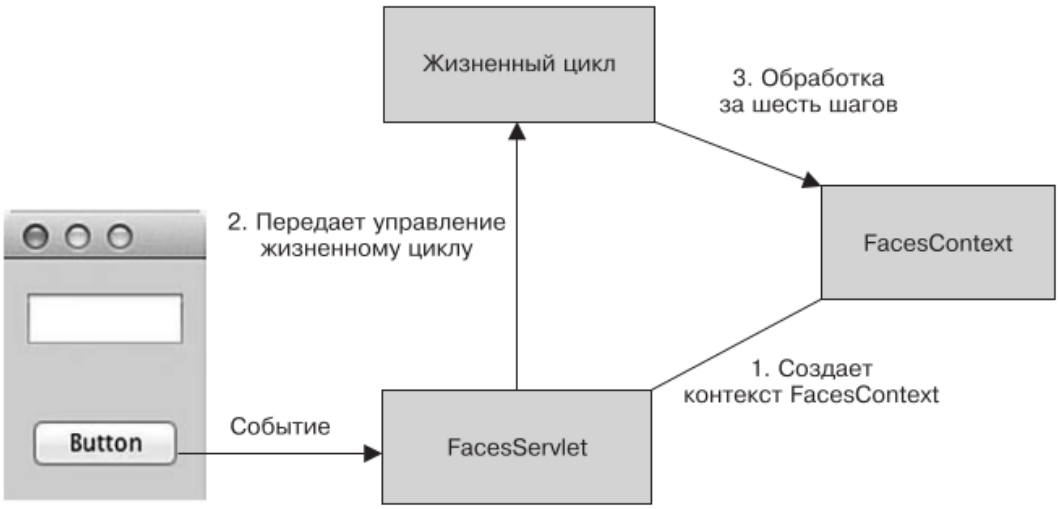

Рисунок 2.13 — Схема обработки запроса сервлетом [17]

При запуске сервера приложений запускается и приложение построенное на основе компоненты JSF. Соответственно, Web-контейнер создает объект типа ***javax.faces.webapp.FacesServlet***, который для приема запросов от браузеров обслуживается собственным листенером.

Отдельный запрос браузера, например, запрос о событии нажатия кнопки, как показано на рисунке 2.13, воспринимается объектом, показанным как ***FacesServlet***, который создает два новых объекта:

1. Объект типа ***javax.faces.context.FacesContext*** для представления контекстной информации, связанной с обработкой входящего запроса и пос ледующего создания соответствующего ответа.
2. Объект типа ***javax.faces.lifecycle.Lifecycle*** для обработки запроса в последовательности шести этапов, показанных на рисунке 2.14.

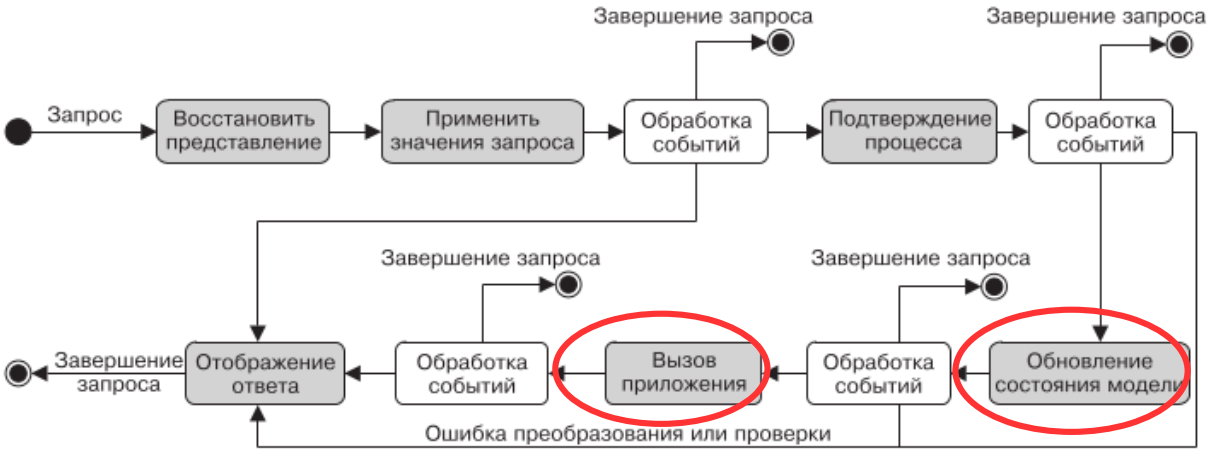

Рисунок 2.14 — Жизненный цикл обработки обработки запроса [17]

В процессе этапов жизненного цикла предусмотнены четыре точки обработки событий, способных досрочно завершить обработку запроса, если отсутствует необходимость в работе последующих этапов. В случае обнаружения ошибки, в каждой точке обработки событий предусмотрено аварийное завершение запроса.

Особое внимание следует уделить этапам «*Обновление состояния модели*» и «*Вызов приложения*».

Обновление состояния модели — этап, связывающий все проверенные значения компонентов с соответствующими компонентами-подложками.

Вызов приложения — этап, чтобы выполнить бизнес-логику. Здесь любое выбранное действие будет выполнено компонентом-подложкой, а также вступает в действие навигация (переход на другие страницы web-приложения).

2.2.2 Контекст состояния запроса FacesContext

Объект типа ***javax.faces.context.FacesContext*** создается сервлетом для представления контекстной информации, связанной с обработкой входящих запросов и создания соответствующего ответа.

Напомню, что общая парадигма программной платформы Java EE — максимальная передача всех служебных или рутинных операций контейнеру сервера приложений, в данном случае и компоненту JSF. Тем не менее, если это необходимо, то программист может воспользоваться информацией, сохраняемой объектом контекста в течении всего жизненного цикла обработки запроса.

Объект контекста создается для каждого запроса браузера. Обычно программист обрабатывает контекст запроса в компоненте-подложки с помощью создания объекта в виде:

```
FacesContext context = FacesContext.getCurrentInstance();
```

К созданному объекту ***context*** можно применить методы, приведенные в таблице 2.1.

Таблица 2.1 — Методы объекта типа FacesContext [17]

Метод	***Описание метода***
addMessage	Присоединяет сообщение: информационное, предупреждающее, сообщение об ошибке, либо сообщение о фатальной ошибке.
getApplication	Возвращает объект типа Application, связанный с этим web-приложением.
getAttributes	Возвращает объект типа Map, представляющий атрибуты, связанные с объектом типа FacesContext.
getCurrentInstance	Возвращает объект типа FacesContext для запроса, который обрабатывается в текущем потоке.
getELContext	Возвращает объект типа ELContext для текущего объекта типа FacesContext.
getMaximumSeverity	Возвращает максимальную степень тяжести, записанную в любом FacesMessage, внесенном в очередь.
getMessages	Возвращает коллекцию объектов типа FacesMessage.
getPartialViewContext	Возвращает объект типа PartialViewContext для заданного запроса. Он используется для внедрения логики в цикл управления обработкой/отрисовкой, например, для обработки запроса AJAX.
getViewRoot	Возвращает корневой компонент, связанный с запросом.

Таблица 2.1 — (продолжение)

Метод	***Описание метода***
release	Высвобождает любые ресурсы, связанные с объектом типа FacesContext.
renderResponse	Сигнализирует реализации JSF о том, что текущая фаза обрабатывающего запросы жизненного цикла была закончена, управление должно быть передано фазе «Отрисовать ответ», минуя любые фазы, которые еще не были выполнены.
responseComplete	Сигнализирует реализации JSF о том, что HTTP-ответ для этого запроса уже был сгенерирован, например, переадресация HTTP, а также о том, что жизненный цикл обработки запросов должен прекратить свою работу, как только завершится текущая фаза.

2.2.3 Модель в виде компонентов-подложек

Компонент-подложка — это ***объект*** аннотированного, в соотвествии с требованиями CDI, JavaBeans-класса (POJO-класса), реализующий бизнес-модель запроса.

Прежде всего заметим, что термин компонент в данной главе и далее понимается в различных семантических значениях:

а) термин **компонет JSF** — обозначение программного модуля, понимаемого в масштабе платформы Java EE (см. подраздел 1.4 первой главы), или как конкретная прикладная библиотека (см. далее пункт 2.2.5);

б) термин **компонет-подложка** — обозначает отдельный JAVA-класс, где обозначение POJO — «*Plain Old Java Object*» или «*Простой Java-объект в старом стиле*»;

в) термин **компонет Facelets** (см. рисунок 2.11 и пункт 2.2.4) — обозначает XHTML-файл с тегами специального формата.

Каждый компонент-подложка имеет свою ***область действия***.

Область действия — это время жизненного цикла объекта в масштабе функционирования приложения, использующего JSF.

Жизненый цикл компонентов подложек, в общем случае, не совпадает с жизненным циклом запросов, которые они обслуживают.

В таблице 2.2 представлены наиболее значимые области действия компонентов-подложек и соотвествующие ис аннотации.

Таблица 2.2 — Области действия компонентов-подложек [17]

Область действия/ Аннотация	***Описание***
Приложение @ApplicationScoped	Наименее ограничительный вариант с самой большой продолжительностью жизни. Созданные объекты доступны во всех циклах запросов/ответов для всех клиентов, использующих веб-приложение, до тех пор, пока приложение активно. Эти объекты можно вызывать одновременно из нескольких источников, они должны быть потокобезопасными. Объекты с такой областью действия могут использовать другие объекты либо без области действия, либо с такой же областью действия.
Сессия @SessionScoped	Объекты доступны для любых циклов запросов/ответов, которые принадлежат сессии клиента. Эти объекты имеют состояние, которое сохраняется между запросами и хранится до тех пор, пока сессия не будет завершена. Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения.
Представление @ViewScoped	Объекты доступны в пределах заданного представления, пока оно не изменится, и их состояние сохраняется до тех пор, пока пользователь не перейдет к новому представлению (в этот момент состояние стирается). Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения или сеанса.
Запрос @RequestScoped	Объект доступен в начале запроса и до тех пор, пока клиенту не был отправлен ответ. Клиент может выполнять несколько запросов, но остаться на одном и том же представлении. Поэтому продолжительность @ViewScoped больше, чем @RequestScoped. Объекты с такой областью действия могут использовать и другие объекты либо без области действия, либо с такой же областью действия, либо с областью действия на уровне приложения или сеанса.
Поток @FlowScoped	Объекты в этой области действия создаются, когда пользователь входит в указанный поток, и освобождаются, когда он выходит из потока.

Чтобы быть управляемым компонентом CDI, компонент-подложка должен иметь аннотацию ***@Named***.

Жизненный цикл компонента-подложки можно контролировать с помощью аннотаций ***@PostConstruct*** и ***@PreDestroy***.

На рисунке 2.15 показана схема жизненного цикла любого компонента-подложки, который может быть аннотирован следующим образом:

а) если перед методом компонента поставить аннотацию ***@PostContruct***, то метод будет вызван, когда объект компонента будет создан и готов к использованию;

б) если перед методом компонента поставить аннотацию ***@PreDestroy***, то метод будет вызван, перед тем как объект компонента будет уничтожен-контейнером.

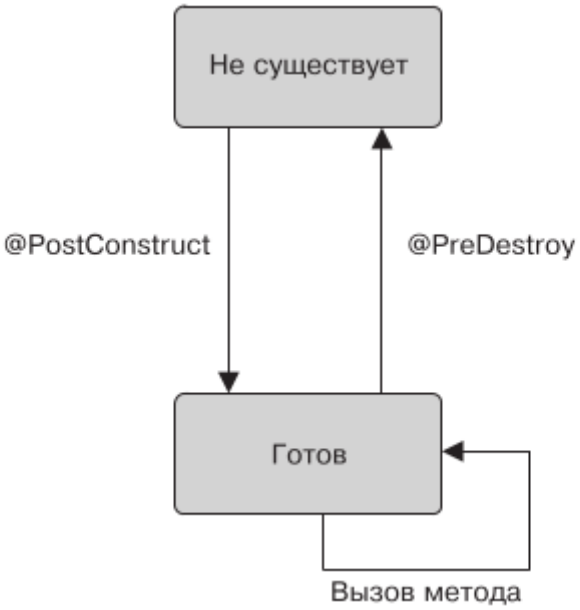

Рисунок 2.15 — Схема жизненного цикла компонента-подложки [17]

Для демонстрации примера компоненты-подложки обратимся к бизнес-модели тестового примера:

1) создадим в Eclipse новый проект типа Dynamic Web Project с именем ***labs***;
2) в этом проекте создадим JAVA-класс ***asu.rsos.TestTomee.java***, содержимое которого показано на листинге 2.1.

Листинг 2.1 — Компонента-подложка TestTomee.java проекта labs

```
package asu.rsos;

import java.util.Date;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named
@ApplicationScoped
public class TestTomee
```

```
{
    // Разделяемый ресурс
    private List<String> msgs = new ArrayList<>();

    // Строка ввода нового сообщения
    private String text = "";

    /**
      * Пустой конструктор
      */
    public TestTomee() {}

    /**
     * Стандартный набор методов Java Beans
     * с особыми правилами именования методов
     */
    public String getMsgs() { return msgs; }

    /**
     * Список msgs - полько для чтения
    public void setMsgs(List<String> msgs)
    {
      this.msgs = msgs;
    }
     */

    /**
     * Методы чтения и установки текста
     */
    public String getText() {return "";}

    public void setText(String text) {this.text = text;}

    /**
     * Метод добавления нового сообщения.
     * Возвращает:
     * - адрес нового ресурса;
     * - любое слово, при использовании faces-config.xml;
     * - null, если страница не изменяется.
     */
    public String addMessage()
    {
            Date dt = new Date();
            String s = dt.toLocaleString() + " " + text;
            msgs.add(s);
            return null;
    }
}
```

В целом, ***TestTomee.java*** в таком написании соответствует структуре POJO-класса, в котором методы ***get*()*** и ***set*(...)*** предназначены для работы с приватными данными. В дальнейшем, такое описание будет автоматически обрабатываться компонентой JSF. В частности, приватная переменная ***msgs*** будет доступна «*только для чтения*».

2.2.4 Представление (View) средствами Facelets

JSF — полностью серверная технология, в которой за представление отвечает «*механизм*» Facelets.

Как было показано на рисунке 2.11 (см. стр. 70), в компоненте JSF за представления могут отвечать различные технологии:

а) **XUL** (*XML User Interface Language*) — язык разметки для создания динамических пользовательских интерфейсов на основе языка XML, разрабатываемый в рамках проекта ***Mozilla***;

б) **JSP** (*JavaServer Pages*) — язык разметки для создания динамических пользовательских интерфейсов, разрабатываемый Eclipse Foundation по стандарту ***JSR 245***;

в) **Facelets** — язык разметки для создания динамических пользовательских интерфейсов, разработанный компанией Sun Microsystems специально для проекта ***JSF***.

Facelets передает разметку ***Отрисовщику***, который отвечает за отображение компонента и перевод пользовательского ввода в значение компонента.

PDL (*Page Description Language*) технологии Facelets построен на основе языка XML и отображается в проектах в виде XHTML-файлов, используя теги из пространства имен, представленных в таблице 2.3.

Таблица 2.3 — Библиотеки тегов, используемые Facelets PDL [17]

URI	***Префикс***	***Описание***
http://xmlns.jcp.org/jsf/html	h	Библиотека классов содержит компоненты и их HTML-отрисовки — h:commandButton, h:comand Link, h:inputText и т. д.
http://xmlns.jcp.org/jsf/core	f	Библиотека содержит пользовательские дейст-вия, которые независимы от созданных ранее отрисовок — f:selectItem, f:validateLength, f:convert Number и т. д.
http://xmlns.jcp.org/jsf/facelets	ui	Теги этой библиотеки добавляют поддержку шаблонов.
http://xmlns.jcp.org/jsf/composite	composite	Библиотека тегов применяется для объявления и определения составных компонентов.
http://xmlns.jcp.org/jsp/jstl/core	c	Facelets-страницы могут использовать некоторые основные библиотеки тегов JSP — <c:if/>, <c:forEach/> или <c:catch/>.
http://xmlns.jcp.org/jsp/jstl/functions	fn	Facelets-страницы могут использовать все функции библиотек тегов JSP.

Указанные в таблице библиотеки тегов (красным цветом выделены ис-

пользуемые далее) отображаются в заголовках XHTML-файлов. Например, если нам будут нужны теги с префиксами ***h*** и ***f***, то заголовок файла будет иметь вид показанный на рисунке 2.16.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
```

Рисунок 2.16 — Заголовок XHTML-файла, использующего теги h и f [17]

«*Механизм*» Facelets управляет шаблонами компоненты JSF.

Типичное современное Web-приложение имеет представление содержащее сразу несколько HTML/XHTML-страниц, часть из которых или все представляют статические (неизменяемые) страницы. Facelets учитывает их и обновляет только динамические страницы. Для примера, рассмотрим типовой набор из пяти HTML-страниц, показанный на рисунке 2.17.

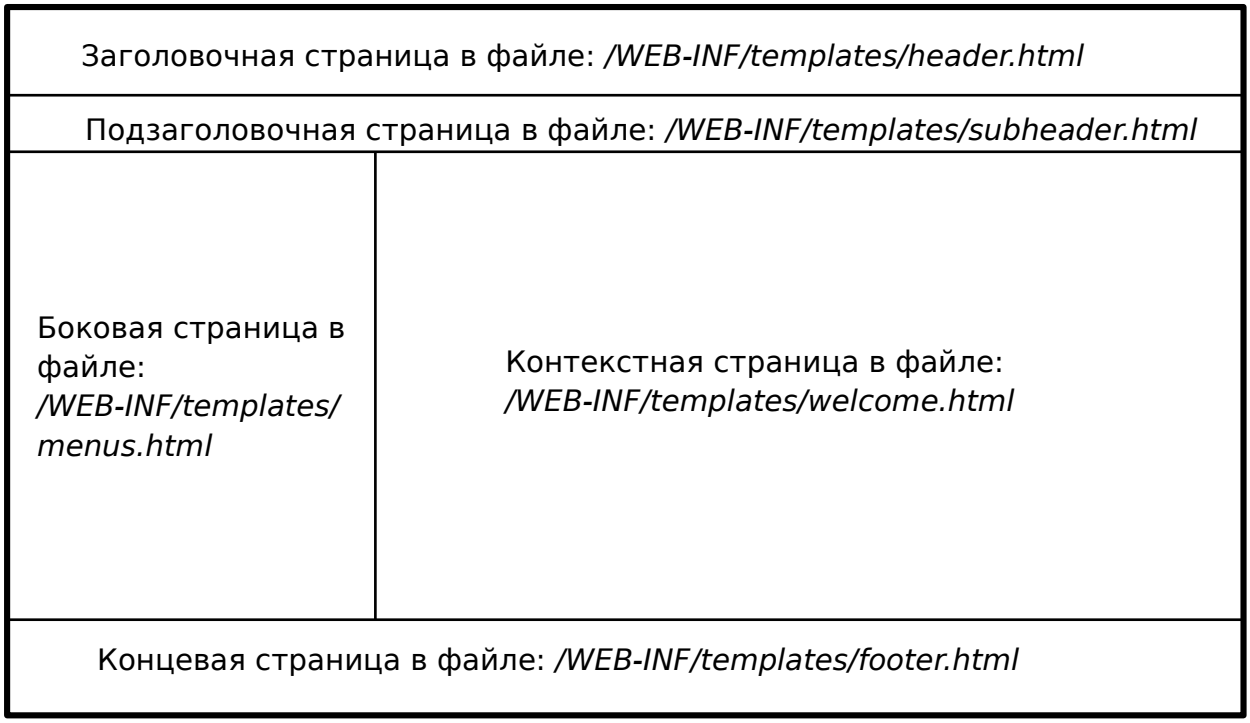

Рисунок 2.17 — Шаблон представления из пяти HTML-страниц

Создадим Facelets-шаблон ***/WEB-INF/templates/lab2Templ.xhtml***, предполагая, что он включает все файлы, показанные на рисунке 2.17. Дополнительно

предполагается, что мы используем теги компоновки, представленные в таблице 2.4. Результат такого шаблона представлен на листинге 2.2.

Таблица 2.4 — Теги Facelets-шаблонов [17]

Тег	***Описание тега***
<ui:composition>	Определяет композицию, которая может использовать шаблон. Несколько композиций могут применять один и тот же шаблон.
<ui:component>	Создает компонент.
<ui:debug>	Собирает информацию об отладке.
<ui:define>	Определяет содержимое, которое вставлено на страницу с помощью шаблона.
<ui:decorate>	Позволяет декорировать содержимое на странице.
<ui:fragment>	Добавляет фрагмент страницы.
<ui:include>	Инкапсулирует и повторно использует содержимое среди нескольких страниц XHTML.
<ui:insert>	Вставляет содержимое в шаблон.
<ui:param>	Передает параметры во включенный файл с использованием тега <ui:include> или шаблона.
<ui:repeat>	Проходит по всему списку объектов.
<ui:remove>	Удаляет содержимое страницы.

Листинг 2.2 — Шаблон lab2Templ.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title>Проект labs</title>
</head>

<body>

      <div id="header">
      <ui:insert name="header">
            <ui:include src="/WEB-INF/templates/header.html"/>
      </ui:insert>
      </div>

      <div id="subheader">
      <ui:insert name="subheader">
```

```
            <ui:include src="/WEB-INF/templates/subheader.html"/>
      </ui:insert>
      </div>

      <table width="100%" border="0" cellpadding="0" cellspacing="0">
    <tr>
    <td id="menus" valign="top" style="width: 25%;">

      <ui:insert name="menus">
              <ui:include src="/WEB-INF/templates/menus.html"/>
      </ui:insert>
      </td>

    <td id="context" valign="top">

      <ui:insert name="content">
              <ui:include src="/WEB-INF/templates/welcome.html"/>
      </ui:insert>
    </td>
    </tr>
    </table>

      <div id="footer">
      <ui:insert name="footer">
              <ui:include src="/WEB-INF/templates/footer.html"/>
            </ui:insert>
      </div>

      </body>
</html>
```

Далее, следует создать файлы HTML-страниц, на которые ссылается бозовый шаблон ***lab2Templ.xhtml***. Если следовать представлению рисунка 2.17, то содержимое этих файлов — достаточно примитивно, поэтому мы не будем анализировать их листинги. Но для демонстрации примера использования шаблона, создадим файл ресурса представления ***/index.xhtml***, содержимое которого показано на листинге 2.3.

Листинг 2.3 — Содержимое файла index.xhtml проекта labs

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

      <ui:composition template="/WEB-INF/templates/lab2Templ.xhtml">
      </ui:composition>
</html>
```

2.2.5 JSF OmniFaces

Компонента JSF платформы Java EE имеет множество реализаций. В данном учебном пособии используется проект ***OmniFaces***.

В прикладном плане, наш проект ***labs*** готов для запуска первых демонстрационных примеров, но к нему необходимо подключить конкретную реализацию компонента JSF. Для этого, формально, имеются две реализации JSF, претендующие на роль эталонных решений:

а) **проект Mojarra** с открытым исходным кодом от компании Oracle, доступный в сервере приложений ***GlassFish***;

б) **проект MyFaces**, который разрабатывается и продвигается организацией-фондом Apache Software Foundation, доступный в частности в сервере приложений ***Apache TomEE***.

Тем не менее, в связи с текущим состоянием дел и переменами, которые обсуждались в подразделе 1.5 первой главы, последняя версия JSF 2.3 проекта MyFaces не может быть использована в учебном процессе, поэтому далее используется сторонний проект ***OmniFaces*** с текущей версией 3.7.1 [25].

Данный пункт учебного пособия описывает последовательность шагов, достаточных для преобразования проекта ***Dynamic Web Project*** в проект обеспечивающий функционирование компонента JSF OmniFaces.

Подготовка проекта требует выполнения всего четырех операций, которые также демонстрируются рисунком 2.18:

1) положить файл ***omnifaces-3.7.1.jar***, представляющий библиотеку JSF, в каталог ***WebContent/WEB-INF/lib/*** проекта (выделено красным цветом);
2) создать в каталоге ***WebContent/META-INF/*** файл ***context.xml***, содержимое которого приведено на листинге 2.4 (выделено синим цветом);
3) создать в каталоге ***WebContent/WEB-INF/*** файл ***beans.xml***, содержимое которого приведено на листинге 2.5 (выделено зеленым цветом);
4) модифицировать дескриптор развертывания проекта (файл ***web.xml***), как это показано на листинге 2.6.

Указанные четыре преобразования являются достаточными и могут быть применены к любому новому проекту , *но не к проекту HttpServlet* !

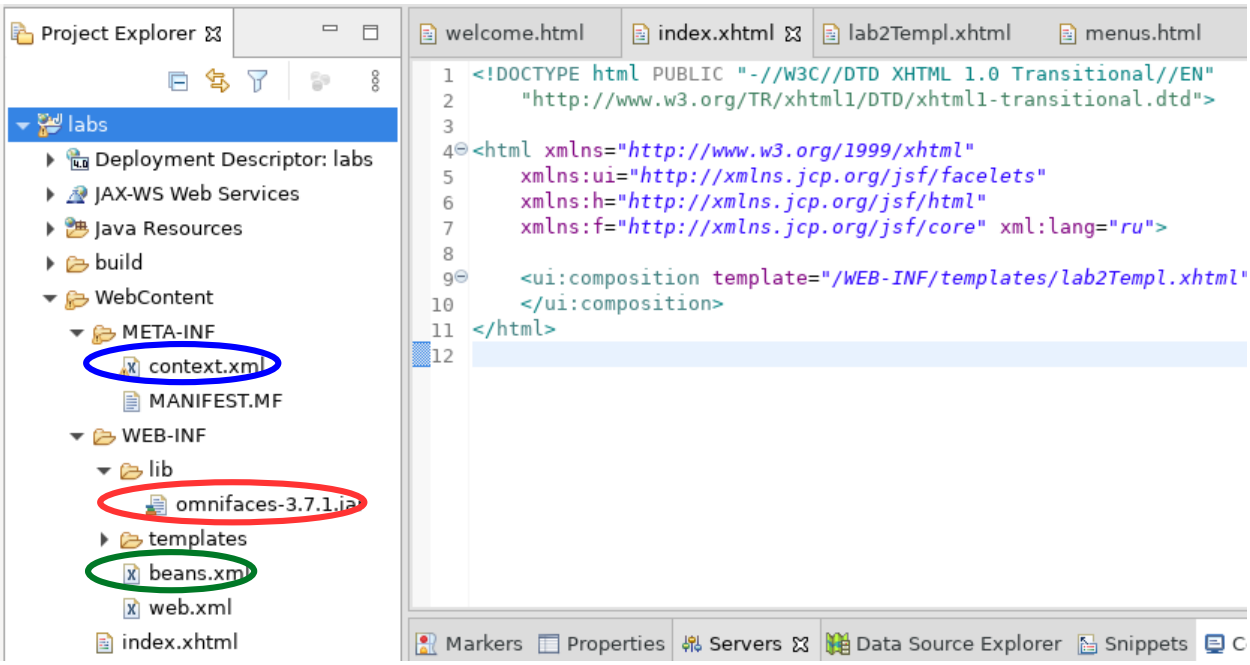

Рисунок 2.18 — Установка OmniFaces и файлов конфигурации в проекте labs

Файл ***omnifaces-3.7.1.jar*** — это последняя на момент написания пособия версия библиотеки JSF. Она реализована так, что может находиться только в указанном красным цветом месте проекта. В частости, любые библиотеки помещенные в этот каталог будут автоматически доступны проекту.

Листинг 2.4 — Дескриптор ресурсов context.xml проекта labs

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Resource name="BeanManager"
        auth="Container"
        type="javax.enterprise.inject.spi.BeanManager"
        factory="org.apache.webbeans.container.ManagerObjectFactory" />
</Context>
```

Файл ***context.xml*** — минимально необходимый для JSF дескриптор ресурсов сервера Apache Tomcat. В данном случае, указывается ресурс менеджера ***BeanManadger***, который управляет прикладными компонентами проекта. В дальнейшем этот файл может использоваться для добавления других ресурсов необходимых проекту, например, ресурсов используемых баз данных.

Листинг 2.5 — Дескриптор развертывания CDI beans.xml проекта labs

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
```

```
       http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd"
       bean-discovery-mode="all" version="2.0">
</beans>
```

Файл ***beans.xml*** — дескриптор развертывания управляемых компонентов CDI. Необходимость в нем отпала, поскольку используются аннотации, но нужен «пустой» файл с содержимым подобным содержимому листинга 2.5.

Листинг 2.6 — Дескриптор приложения web.xml проекта labs

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

<!-- Имя проекта -->
  <display-name>labs</display-name>

<!-- Сервлеты и фильтры. -->

      <servlet>
            <servlet-name>facesServlet</servlet-name>
            <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
            <load-on-startup>1</load-on-startup>
      </servlet>

<!-- Какие web-ресурсы - доступны -->
      <servlet-mapping>
                  <servlet-name>facesServlet</servlet-name>
                  <url-pattern>*.xhtml</url-pattern>
      </servlet-mapping>
      <servlet-mapping>
            <servlet-name>facesServlet</servlet-name>
            <url-pattern>*.html</url-pattern>
      </servlet-mapping>

      <filter>
<filter-name>characterEncodingFilter</filter-name>
<filter-class>org.omnifaces.filter.CharacterEncodingFilter</filter-class>
      </filter>
      <filter-mapping>
            <filter-name>characterEncodingFilter</filter-name>
            <servlet-name>facesServlet</servlet-name>
      </filter-mapping>

      <filter>
      <filter-name>gzipResponseFilter</filter-name>
      <filter-class>org.omnifaces.filter.GzipResponseFilter</filter-class>
      </filter>
      <filter-mapping>
            <filter-name>gzipResponseFilter</filter-name>
            <servlet-name>facesServlet</servlet-name>
            <dispatcher>REQUEST</dispatcher>
```

```
            <dispatcher>ERROR</dispatcher>
        </filter-mapping>

        <filter>
        <filter-name>noCacheFilter</filter-name>
        <filter-class>org.omnifaces.filter.CacheControlFilter</filter-class>
        </filter>
        <filter-mapping>
            <filter-name>noCacheFilter</filter-name>
            <servlet-name>facesServlet</servlet-name>
        </filter-mapping>

        <filter>
        <filter-name>facesExceptionFilter</filter-name>
        <filter-class>org.omnifaces.filter.FacesExceptionFilter</filter-class>
        </filter>
        <filter-mapping>
            <filter-name>facesExceptionFilter</filter-name>
            <servlet-name>facesServlet</servlet-name>
        </filter-mapping>

<!-- Листенеры - те, кто принимает какие-либо запросы -->
  <listener>
    <listener-class>org.apache.webbeans.servlet.WebBeansConfigurationListener
    </listener-class>
  </listener>

<!-- Ресурс, который может быть доступен по умолчанию -->
  <welcome-file-list>
    <welcome-file>default.xhtml</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Файл ***web.xml*** — главный дескриптор развертывания для проекта ***labs***. В нем, в частности, описываются ресурсы, которые не были указаны в файле ***context.xml***.

Выше, на листинге 2.6, указаны только ресурсы, необходимые для запуска JSF ***OmniFaces*** и достаточные для запуска наших демострационных примеров данного пункта. Когда наш проект будет развиваться, то среда разработки Eclipse EE может добавить в этот файл описание ресурсов, коорые необходимы дополнительно.

Если теперь запустить сервер приложений с проектом ***labs*** и обратиться к URI-ресурсу http://localhost:8080/labs/index.xhtml, то мы получим изображение шаблона рисунка 2.17, в виде показанном на рисунке 2.19.

Обратите внимание, что рисунки 2.17 и 2.19 отличаются только стилем отображения, но имеют общую геометрическую архитектуру размещения своих компонент. Если запустить любой браузер и обратиться к URI-ресурсу http://localhost:8080/labs/, то изображение шаблона не изменится.

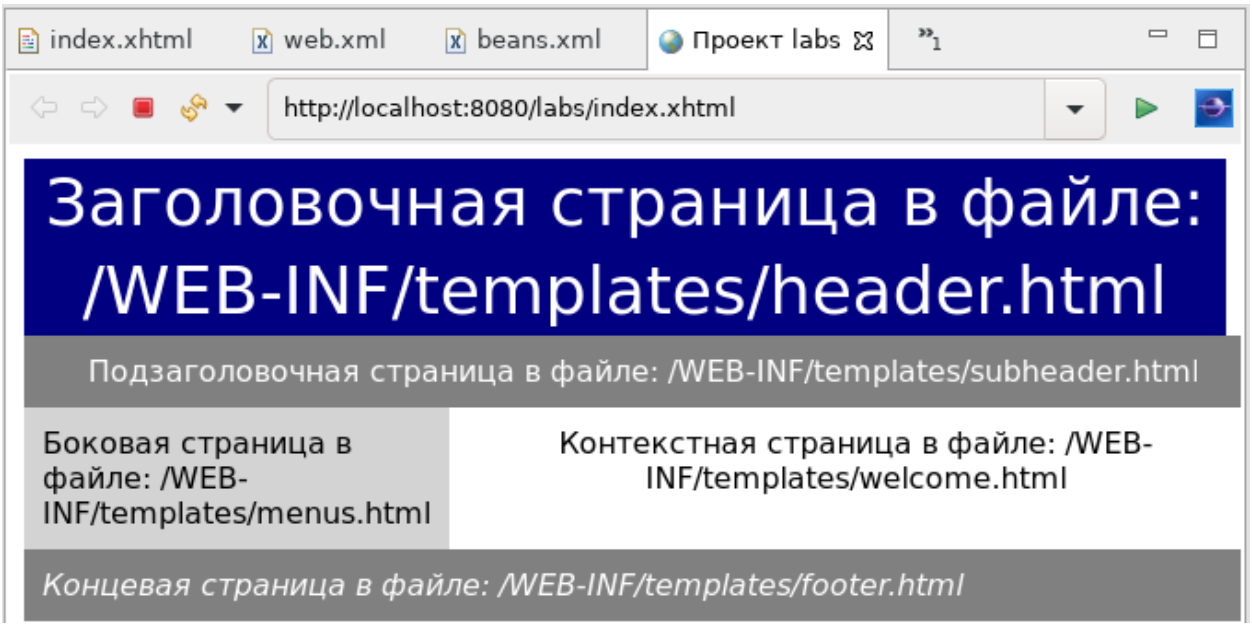

Рисунок 2.19 — Изображение Facelets-шаблона /WEB-INF/templates/lab2Templ.xhtml

Каждый шаблон допускает множество вариантов использования.

Каждый Facelets-шаблон может использоваться для различных бизнес-приложений. В этом и состоит основная парадигма программной платформы Java EE. Для этого используются теги, представленные ранее в таблице 2.4 . Например, тег ***<ui:define>*** переопределяет компоненту шаблона по его имени. Если изменить содержимое файла ***index.xhtml***, как это показано на листинге 2.7, то изменится и вид подзаголочной страницы, а боковая страница будет удалена.

Листинг 2.7 — Содержимое файла index.xhtml проекта labs

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

      <ui:composition template="/WEB-INF/templates/lab2Templ.xhtml">
            <ui:define name="subheader">Новое содержание</ui:define>
            <ui:define name="menus"></ui:define>
</ui:composition>
</html>
```

2.3 Реализация тестового примера средствами JSF

Учебная цель данного подраздела — изучить базовые средства практического использования компоненты JSF.

Важность практического изучения возможностей JSF связана с явным подобием ее функциональной архитектуры, видимой клиентом Web-приложения и бизнес-парадигмой модели SOA, представленной в предыдущей главе на рисунке 1.12 (см. стр. 32). Действительно, клиент JSF-приложения обращается к Web-приложению посредством указания URI-ресурса в виде файла XHTML, который (на основе Facelets-шаблона) отражает «***Представление***», скрытого за ним, бизнес-приложения. Аналогично, бизнес-руководитель предприятия к бизнес-функции уровня приложений, «***Представлению***» уровня бизнес-логики. Таким образом, технология Web-приложений на основе компоненты JSF — явный кандидат на технологию реализации уровня бизнес-логики предприятия.

Если рассматривать учебный процесс данной дисциплины в интерпретации функционирования предприятия, то учебный материал данного подраздела — это формирование и реализация уровня бизнес-логики такого предприятия.

2.3.1 Создание Facelets-шаблона изучаемой дисциплины

Базовым ресурсом Facelets-компонент являются XHTML-файлы.

Новый шаблон Facelets-компонент создадим по общей схеме предыдущего шаблона, как это приведено в таблице 2.5.

Обратите внимание, что:

а) ***все компоненты*** шаблона представлены файлами в XHTML-формате;

б) ***имена компонент*** шаблона отличаются от имен старого шаблона только цифрой и располагаются в тех же каталогах;

в) ***зеленым цветом*** выделены статические компоненты, не требующие новых знаний;

г) ***синим цветом*** выделены компоненты, использующие ссылки;

д) ***красным цветом*** выделена компонента, использующая список.

Таблица 2.5 - Facelets-компонент нового шаблона

Компонента	***Описание***
lab3Templ.xhtml	Базовый шаблон. Листинг 2.8.
header3.xhtml	Заголовок учебной дисциплины. Листинг 2.9.
subheader3.xhtml	Вид занятия. Листинг 2.10.
footer3.xhtml	Концевик приложения. Листинг 2.11.
welcome3.xhtml	Контекст приложения. Основной перезагружаемый компонент. Листинг 2.12.
menus3.xhtml	Компонента боковой страницы. Листинг 2.13.
lab3.xhtml	Дополнительный ресурс. Листинг 2.14.
auth3.xhtml	Дополнительный ресурс. Листинг 2.15.
default.xhtml	Доступный ресурс по умолчанию. Листинг 2.16.

Листинг 2.8 — Базовый шаблон lab3Templ.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title>Новый шаблон labs</title>
</head>

<body>

      <div id="header">
      <ui:insert name="header">
            <ui:include src="/WEB-INF/templates/header3.xhtml"/>
      </ui:insert>
      </div>

      <div id="subheader">
      <ui:insert name="subheader">
            <ui:include src="/WEB-INF/templates/subheader3.xhtml"/>
      </ui:insert>
      </div>

      <table width="100%" border="0" cellpadding="0" cellspacing="0">
      <tr>
      <td id="menus" valign="top" width="1px">

      <ui:insert name="menus">
             <ui:include src="/WEB-INF/templates/menus3.xhtml"/>
      </ui:insert>
      </td>
```

```
    <td id="context" valign="top">

      <ui:insert name="content">
              <ui:include src="/WEB-INF/templates/welcome3.xhtml"/>
      </ui:insert>
    </td>
    </tr>
    </table>

      <div id="footer">
      <ui:insert name="footer">
              <ui:include src="/WEB-INF/templates/footer3.xhtml"/>
            </ui:insert>
      </div>

      </body>
</html>
```

В листинге 2.8 комментировать нечего, поскольку в нем изменены только ссылки на компоненты, согласно именам таблицы 2.5.

Листинг 2.9 — Заголовок учебной дисциплины header3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Заголовок</title>
</head>
<body>
       <div  align="center" style="width:100%;font-size:26px;line-height:28px;
                  background-color:navy;color:white;padding:10px">
                  Распределенные сервис-ориентированные системы
      </div>
</body>
</html>
```

В листинге 2.9 изменен только текст и размер шрифта.

Листинг 2.10 — Вид занятия subheader3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Вид занятий</title>
</head>
<body>
            <div  style="background-color:gray;width:100%;
```

```
                        color:white;padding:5px">
                        Лабораторные работы
      </div>
</body>
</html>
```

В листинге 2.10 изменен только текст.

Для навигации между ресурсами без помощи компонент-подложек используются теги-целей из библиотеки тегов http://xmlns.jcp.org/jsf/html (см. таблицу 2.3 на стр. 79), которые определенны в таблице 2.6.

Таблица 2.6 — Теги целей [17]

Тег	***Описание***
<h:button>	Отрисовывает элемент ввода данных HTML для кнопки, при нажатии на которую создается HTTP-запрос GET.
<h:link>	Отрисовывает элемент привязки HTML <a>, при нажатии на которую создается HTTP-запрос GET.

Формат использования этих тегов — смотри в листингах 2.11 и 2.12.

Листинг 2.11 — Концевик приложения footer3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Концевик приложения</title>
</head>
<body>
            <div style="background-color:gray;width:100%;color:white;
                        padding:20px;font-style:italic">

             Учебный программный комплекс "УПК АСУ" <br/>
             Преподаватель: В.Г. Резник <br/>
             <h:link outcome="default.xhtml" value="Перейти на главную..."/>

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

Листинг 2.12 — Контекст приложения welcome3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Insert title here</title>
</head>
<body>
       <div align="center" style="color:black; width:100%; padding:10px">
                  <h2>Приветствуем Вас на сайте дисциплины РСОС!</h2>

                  <p><a href="http://localhost:8080/test/TestTomee">
                              Запуск тестового примера...</a></p>

                  <p><h:link outcome="index.xhtml"
                              value="Показать базовый шаблон..."/> </p>

                  <p><h:link outcome="lab3.xhtml"
                              value="Лабораторная работа №3"/> </p>

                  <p><h:button outcome="auth3.xhtml"
                              value="На авторизацию..."/> </p>
      </div>
</body>
</html>
```

Для удобного ввода информации используется набор ***тегов-выбора***, основанных на двух базовых библиотеках тегов http://xmlns.jcp.org/jsf/html и http://xmlns.jcp.org/jsf/core.

Перечень указанных тегов-выбора представлен в таблице 2.7.

Таблица 2.7 — Теги-выбора [17]

Тег	***Описание***
<h:selectBooleanCheckbox>	Отображается один флажок, представляющий двоичное значение. Флажок будет установлен в зависимости от значения свойства.
<h:selectManyCheckbox>	Отрисовывается список флажков.
<h:selectManyListbox>	Отрисовывается компонент множественного выбора, где могут быть выбраны один или несколько вариантов.
<h:selectManyMenu>	Отрисовывается HTML-элемент <select>.
<h:selectOneListbox>	Отрисовывается компонент одиночного выбора, где может быть выбран только один вариант.
<h:selectOneMenu>	Отрисовывается компонент одиночного выбора, где может быть выбран только один вариант. Показывает только один доступный вариант в любой момент времени.
<h:selectOneRadio>	Отрисовывает список переключателей.

Следует заметить, что теги-выбора, имея стилизованное графическое представление, предназначены для ввода бизнес-информации и для нормального использования привязывается к компоненте-подложке (бизнес-модели).

В данном пункте, мы создаем Facelets-шаблон изучаемой дисциплины или, другими словами, - уровень бизнес-логики предприятия, который еще не привязан к бизнес модели.

Далее, на листинге 2.13 представлена компонента ***menus.xhtml***, которая в декоративной форме выводит список лабораторных работ с помощью тега ***<h:selectOneListbox>***.

Листинг 2.13 — Компонента боковой страницы menus3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://xmlns.jcp.org/jsf/html"
        xmlns:f="http://xmlns.jcp.org/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Боковая страница</title>
</head>
<body>
      <div  style="background-color:lightgray;
                  color:black;padding:10px">

                                  <b>Список работ</b> <hr/>

                  <h:selectOneMenu>
                  <f:selectItem itemLabel="Работа №1"/>
                  <f:selectItem itemLabel="Работа №2"/>
                  <f:selectItem itemLabel="Работа №3"/>
                  <f:selectItem itemLabel="Работа №4"/>
                  <f:selectItem itemLabel="Работа №5"/>
                  <f:selectItem itemLabel="Работа №6"/>
                  <f:selectItem itemLabel="Работа №7"/>
                  <f:selectItem itemLabel="Работа №8"/>
                  <f:selectItem itemLabel="Работа №9"/>
                  </h:selectOneMenu>  <hr/>

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

Далее представлены листинги двух ресурсов ***lab3.xhtml*** и ***lab3.xhtml***, которые будут задействованы в следующих пунктах данного подраздела, а здесь участвуют в качестве демонстрационного заполнения.

Листинг 2.14 — Доступный пользователю ресурс lab3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Приложение lab3</title>
</head>
<body>
            <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">
                  <ui:define name="context">
                        Работа lab3 еще не сделана!
                  </ui:define>
            </ui:composition>
</body>
</html>
```

Листинг 2.15 — Доступный пользователю ресурс auth3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Приложение auth3</title>
</head>
<body>
            <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">
                  <ui:define name="context">
                        Работа auth3 еще не сделана!
                  </ui:define>
            </ui:composition>
</body>
</html>
```

Прямой доступ к новому Facelets-шаблону осуществляется с помощью ресурса ***default.xhtml***, представленного на следующем листинге.

Листинг 2.16 — Доступный пользователю ресурс default.xhtml проекта labs

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

            <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">
                  <!--
                        <ui:define name="menus"></ui:define>
             -->
            </ui:composition>
</html>
```

Теперь в рапоряжении имеются все компоненты. Запуск файла ресурса ***default.xhtml*** покажет нам новый Facelets-шаблон с вполне функциональной заставкой приветствия (см. рисунок 2.20).

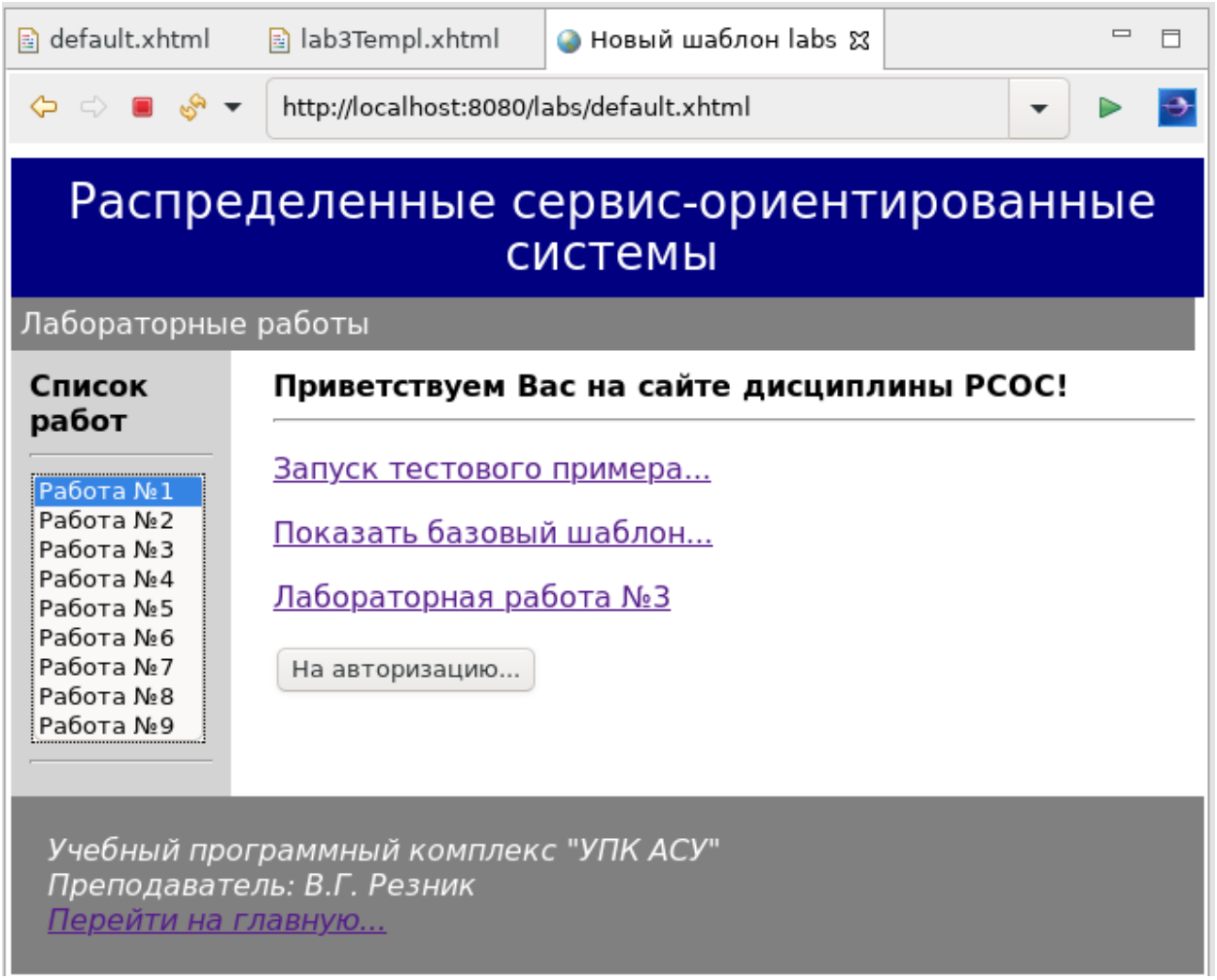

Рисунок 2.20 — Новый Facelets-шаблон учебной дисциплины

Обратите внимание, что, если в листинге 2.16 убрать комментарий, то боковая панель показана не будет.

2.3.2 Прямая реализация тестового примера

Наиболее сильная сторона технологии JSP — прямое взаимодействие ресурсов представления информации с бизнес-уровнем компонент-подложек.

Для прямой реализации тестового приложения, ранее созданного на основе сервлета HttpServlet, имеются все необходимые условия:

а) ***создан*** проект ***labs***, включающий в себя компоненту JSF;

б) ***имеется*** новый Facelets-шаблон учебной дисциплины;

в) ***имеется*** компонента-подложка в виде POJO-класса ***TestTomee***, реализующая бизнес-модель тестового примера (см. листинг 2.1 на стр. 77);

г) ***отсутствует*** только XHTML-ресурс представления решаемой задачи, который бы связывал отображаемое в браузере представление задачи с с бизнес-подложкой модели.

Реализуем недостающую часть приложения тестовой задачи на основе файла ***lab3.xhtml***, который уже включен в структуру шаблона учебной дисциплины (см. листинг 2.14 на стр. 93-94). Сачала запустим тестовое приложение (см. рисунок 2.21) и перечислим графические компоненты, которые должен содержать файл ***lab3.xhtml***.

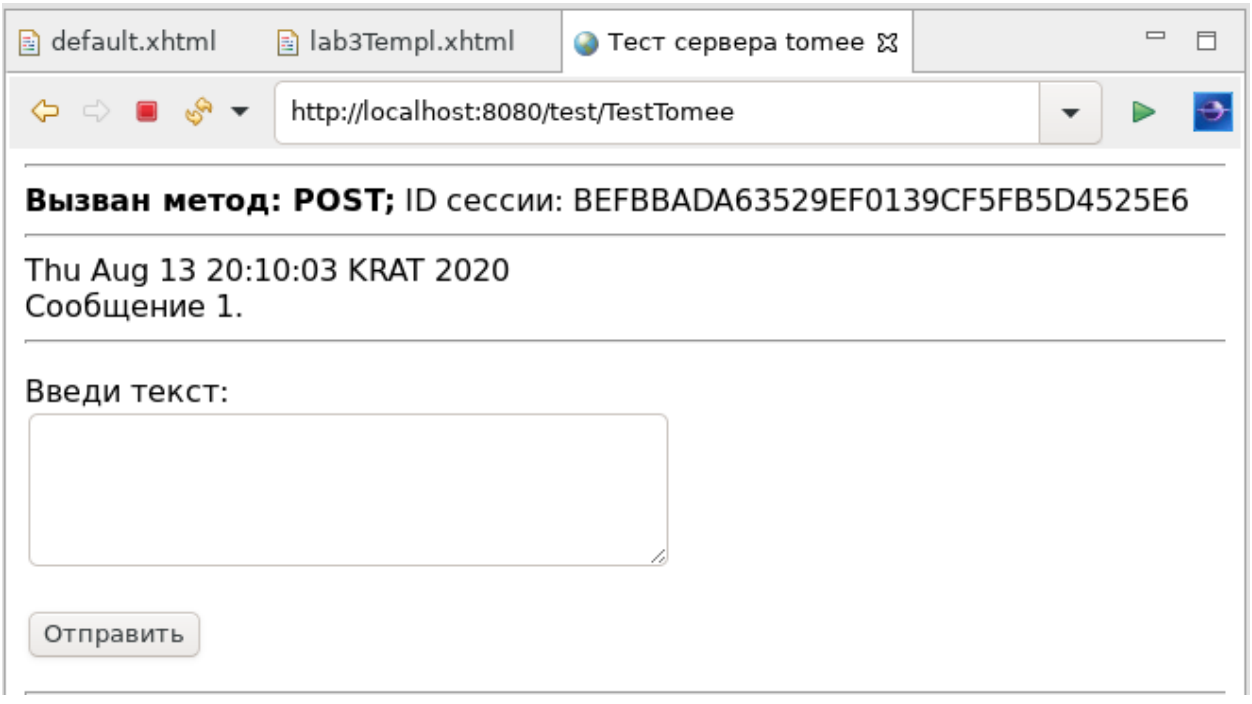

Рисунок 2.21 — Интерфейс тестового приложения

В интерфейсе тестового приложения можно выделить три части, пред-

ставленные в таблице 2.8.

Таблица 2.8 — Основные части интерфейса тестового приложения

№ части	***Описание***
1	Вывод метода обращения к серверу и идентификатора сессии.
2	Вывод списка отправленных приложению сообщений.
3	Форма ввода текста и командная кнопка отправки тестовому приложению нового сообщения.

Хорошо видно, что каждая из выделенных частей приложения требует ввода или вывода информации сосредоточенной в объектах компонентов-подложек или в объектах компоненты JSF.

Проведем последовательную реализацию тестового приложения согласно выделенным в таблице частям, предварительно изучив необходимую для этого теоретическую базу.

Для связывания страниц JSF с компонентами-подложками используется язык выражений ***EL*** (***Expression Language***).

Базовый синтаксис утверждения языка выражений имеет вид ***#{expr}***, который будет оцениваться и преобразовываться во время выполнения JSF.

Основные операторы EL:

а) **арифметические**: **+** , **–** , ***** , / (деление), **%** (целая часть);

б) **операторы отношений**: == (равно), **!=** (не равно), < (меньше), > (больше), <= (меньше либо равно), >= (больше либо равно);

в) **логические**: **&&** (и), || (или), **!** (не);

г) **другие**: **()** (вызов метода), **empty** (логическая проверка на ***null***), **[]** (как и оператор «*точка*» - доступ к атрибуту).

Компоненты-подложки — это CDI-объекты POJO-классов, имеющих имя класса, имена публичных методов и имена приватных данных (простых типов или объектных).

Доступ к целевым приватным данным компонентов-подложек осуществляется через имена классов, причем используется «*верблюжья нотация*», требующая, чтобы имя класса указвалось со строчной буквы. Например, компонента-подложка ***TestTomee*** проекта ***labs*** имеет приватную строковую переменную ***text*** (см. листинг 2.1, стр. 77-78):

```
// Строка ввода нового сообщения
private String text = "";
```

Для обращения к этой переменной со страницы JSF следует использовать синтаксис: **"#{testTomee.text}"**.

Обратите внимание, что в приведенном примере класс ***TestTomee*** должен иметь реализацию методов:

- для чтения из text: ***public String getText() {...}***;
- для записи в text: ***public void setText(String text) {...}***.

Страницы JSF имеют доступ ко множеству ***неявных объектов***, представленных в таблице 2.9.

Теперь у нас имеются все необходимые знания для реализации первой части тестового приложения, которая требует вывода метода доступа к серверу в HTTP-запросе и идентификатора сессии, в пределах которой осуществляется запрос.

Поскольку в таблице 2.8 присутствуют неявные объекты ***request*** и ***session***, то воспользуемся их методами для доступа к нужной информации.

Ранее, нужная нам информация извлекалась в тестовом примере (см. пункт 1.5.5 главы 1). Там JSP-страница ***test.jsp*** (см. листинг 1.2, стр. 55) содержала выражения:

```
<hr>
      <b>Вызван метод: <%= request.getMethod() %>;</b>
      ID сессии:       <%= session.getId() %>
<hr>
```

Зная назначение и структуру POJO-классов (JavaBeans) и синтаксис применения языка выражений EL, мы понимаем, что:

а) метод доступа можно получить выражением: **"#{request.method}"**;

б) идентификатор сессии можно получить выражением: **"#{session.id}"**.

Часть 1 тестового приложения реализуется в Facelets-компоненте нового шаблона с идентификатором ***subheader***.

Исходя из этого условия, переопределим компоненту ***subheader*** JSF-страницы ***lab3.xhtml***, как это приведено в листинге 2.17.

Таблица 2.9 — Неявные объекты, доступные на страницах JSF [17]

Неявный объект	***Описание***	***Возвращаемый тип***
application	Представляет среду веб-приложения. Используется для получения конфигурационных параметров приложения.	Object
applicationScope	Преобразует имена глобальных атрибутов приложения в их значения.	Map
component	Указывает на текущий компонент.	UIComponent
cc	Указывает на текущий составной компонент.	UIComponent
cookie	Определяет объект типа Map, содержащий имена cookies (являющиеся ключами) и объекты типа Cookie.	Map
facesContext	Указывает на объект типа FacesContext для текущего запроса.	FacesContext
flash	Представляет объект, использующий флеш.	Object
header	Преобразует имя HTTP-заголовка в значение заголовка типа String.	Map
headerValues	Преобразует имя HTTP-заголовка в набор всех значений заголовка типа String[].	Map
initParam	Преобразует имена параметров инициализации контекста к значениям типа String.	Map
param	Преобразует имена параметров запроса к одному значению параметра типа String.	Map
paramValues	Преобразует имена параметров запроса к набору всех значений параметра типа String[].	Map
request	Представляет объект запроса HTTP.	Object
requestScope	Преобразует имена атрибутов запроса к их значениям.	Map
resource	Представляет объект ресурса.	Object
session	Представляет объект сеанса HTTP.	Object
sessionScope	Преобразует имена атрибутов сеанса к их значениям.	Map
view	Представляет текущее представление.	UIViewRoot
viewScope	Преобразует имена атрибутов представления к их значениям.	Map

Листинг 2.17 — Первая часть реализации lab3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
            xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

      <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">
            <!-- Переопределение подзаголовка subheader-->
      <ui:define name="subheader">
            <!-- Часть 1 реализации тестового примера -->
      <div  style="background-color:gray;width:100%;
            color:white;padding:5px">
            <b>Вызван метод: #{request.method};</b> ID сессии: #{session.id}
      </div>
      </ui:define>

      </ui:composition>
</html>
```

Результат реализации этой части приложения показан на рисунке 2.22. Обратите внимание, что компонента ***context*** не изменилась.

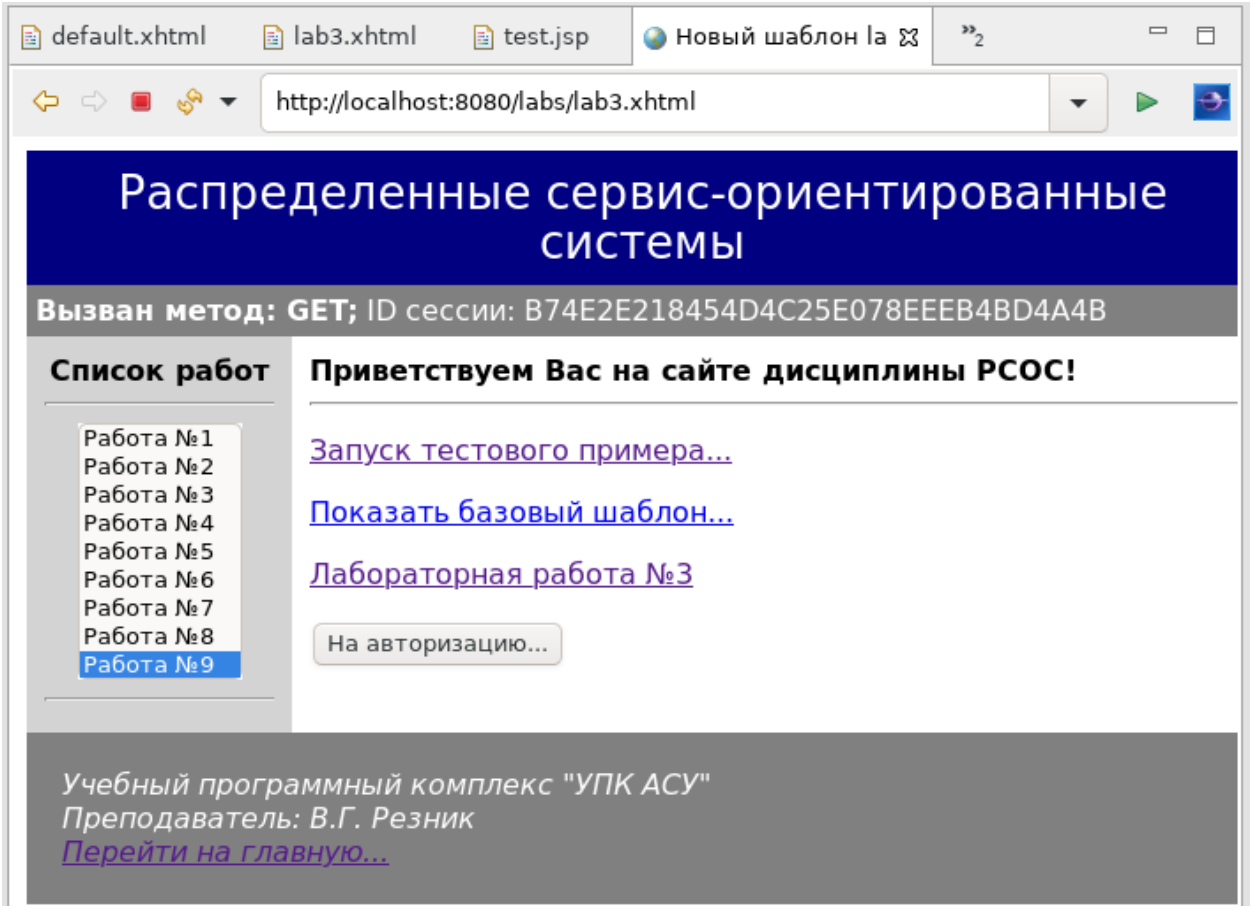

Рисунок 2.22 — Результат реализации первой части тестового приложения

Вторая часть тестового приложения, согласно таблице 2.8, должна выводить список отправленных приложению сообщений. Здесь следует напомнить, что отправленные сообщения храняться в компоненте-подложке класса ***TestTomee*** (см. листинг 2.1 на стр. 77-78) в виде списка строк ***msgs*** типа ***List<String>***. Этот список строк необходимо прочитать на страние ***lab3.xhtml***, а затем представить для обозрения.

JSF имеет четыре основных ***тега вывода***, представленных в таблице 2.10 и способных с помощью EL отображать информцию связанную с компонентами-подложками, а также — ***тег цикла***: <c:forEach>.

Таблица 2.10 — Теги вывода [17]

Тег	***Описание***
<h:outputLabel>	Отображает элемент HTML <label>.
<h:outputLink>	Отображает элемент привязки HTML <a>.
<h:outputText>	Выводит текст.
<h:outputFormat>	Отрисовывает параметризованный текст.

Рассмотрим применение тегов ***<c:forEach>*** и ***<h:outputLabel>*** на примере реализации второй части тестового приложения. Эта часть будет переопределять компоненту шаблона ***context***, что показано на рисуке 2.23.

```
<!-- Переопределение контекстной страницы context -->
<ui:define name="context">
<!-- Часть 2 реализации тестового примера -->
<div align="left" style="color:black;padding:10px">
    <b>Тестовый пример (lab3.xhtml)</b> <hr/>

    <c:forEach var="ss" items="#{testTomee.msgs}">
        <h:outputLabel value="#{ss}"/> <hr/>
    </c:forEach>
</div>
</ui:define>
```

Рисунок 2.23 — Демонстрация использования тегов <c:forEach> и <h:outputLabel>

Результат работы реализованных двух частей тестового приложения показан на рисунке 2.24.

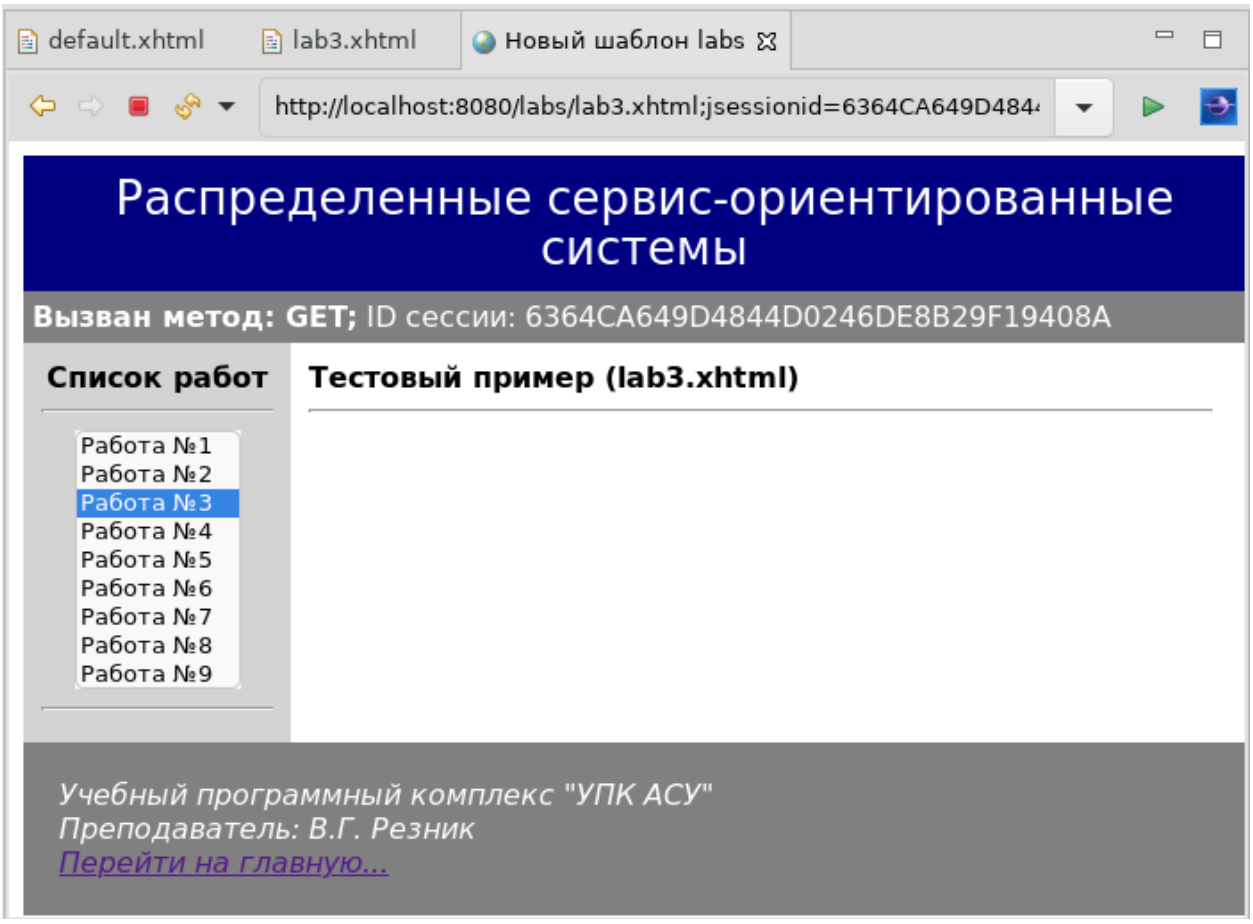

Рисунок 2.24 — Результат работы двух частей тестового приложения

Обратите внимание, что на рисунке 2.24 никаких сообщений не показано, потому что еще не реализованы средства ввода сообщений.

Основными средствами интерактивного взаимодействия JSF-страниц и компонент-подложек являются ***теги ввода*** информации (см. таблица 2.11) и ***теги команд*** (см. таблица 2.12).

Таблица 2.11 — Теги компонентов ввода [17]

Тег	***Описание***
<h:inputHidden>	Представляет HTML-элемент ввода для скрытых данных (полезен для переноса значений от страницы к странице вне сессии).
<h:inputSecret>	Представляет HTML-элемент ввода для паролей. При повторной загрузке страницы любое ранее введенное значение не будет отображено , если только свойство redisplay не имеет значения true.
<h:inputText>	Представляет HTML-элемент ввода для текста.
<h:inputTextarea>	Представляет текстовую область в HTML.
<h:inputFile>	Позволяет просматривать каталог, выбирать и загружать файл.

Таблица 2.12 — Теги команд [17]

Тег	***Описание***
<h:commandButton>	Представляет HTML-элемент, предназначенный для ввода данных кнопок «Отправить» или «Очистить».
<h:commandLink>	Представляет HTML-элемент для гиперссылки, который действует как кнопка «Отправить». Этот компонент должен быть помещен внутрь формы.

Используя выделенные красным цветом теги, можно легко реализовать третью часть тестового приложения, что:

а) тег <h:inputTextarea> читает и записывает приватный атрибут ***text*** класса ***TestTomee***;

б) <h:commandButton> обращается к методу ***addMessage()*** того же класса.

Реализация третьей части приложения оформляется тегом ***<h:form>*** и вставляется в компоненту ***context*** после второй части.

Общий результат реализации приложения в виде файла ***lab3.xhtml*** представлен на листинге 2.18.

Листинг 2.18 — Полная реализация lab3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
            xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

            <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">

            <!-- Переопределение подзаголовка subheader-->
            <ui:define name="subheader">

            <!-- Часть 1 реализации тестового примера -->
            <div  style="background-color:gray;width:100%;
                        color:white;padding:5px">

            <b>Вызван метод: #{request.method};</b> ID сессии: #{session.id}

            </div>

            </ui:define>

            <!-- Переопределение контекстной страницы context -->
            <ui:define name="context">

            <!-- Часть 2 реализации тестового примера -->
```

```
            <div align="left" style="color:black;padding:10px">

            <b>Тестовый пример (lab3.xhtml)</b> <hr/>

            <c:forEach var="ss" items="#{testTomee.msgs}">
                  <h:outputLabel value="#{ss}"/> <hr/>
            </c:forEach>

            <!-- Часть 3 реализации тестового примера -->
            Введи текст: <br/>

            <h:form>
                  <h:inputTextarea rows="5" cols="40"
                        value="#{testTomee.text}"/> <br/>
                  <h:commandButton value="Отправить"
                        action="#{testTomee.addMessage}"/> <hr/>
            </h:form>

            </div>

            </ui:define>

            </ui:composition>

</html>
```

Запуск полной реализации тестового приложения с двумя введенными сообщениями представлен на рисунке 2.25.

Хорошо видно, что с точностью до ряда деталей оформления, прикладная часть тестового JSF-приложения полностью соответствует прикладной части тестового JSP-приложения.

Завершая данный подраздел успешной реализацией тестового приложения, можно достаточно четко сформулировать основные различия между технологиями JSP и JSF.

Технология JSP на первое место выставляет ***HttpServlet***, к которому идут запросы от браузера. Сервлет может сам формировать HTML-страницу или обратиться к соотвествующей JSP-странице. Используя встроенный (неявный) объект ***out***, можно формировать содержимое HTML-страницы, что и было сделано в тестовом примере (см. пункт 1.5.5 главы 1).

Технология JSF на первое место выставляет ***XHTML-ресурс***, к которому обращается браузер, а ***FacesServlet*** «*работает скрытно*», предоставляя каждому запросу браузера объект жизненного цикла (типа ***Lifecycle***) и объект состояния запроса (типа ***FacesContext***). Предоставляемая браузеру прикладная информация выводится не в страницу, а — в поле страницы.

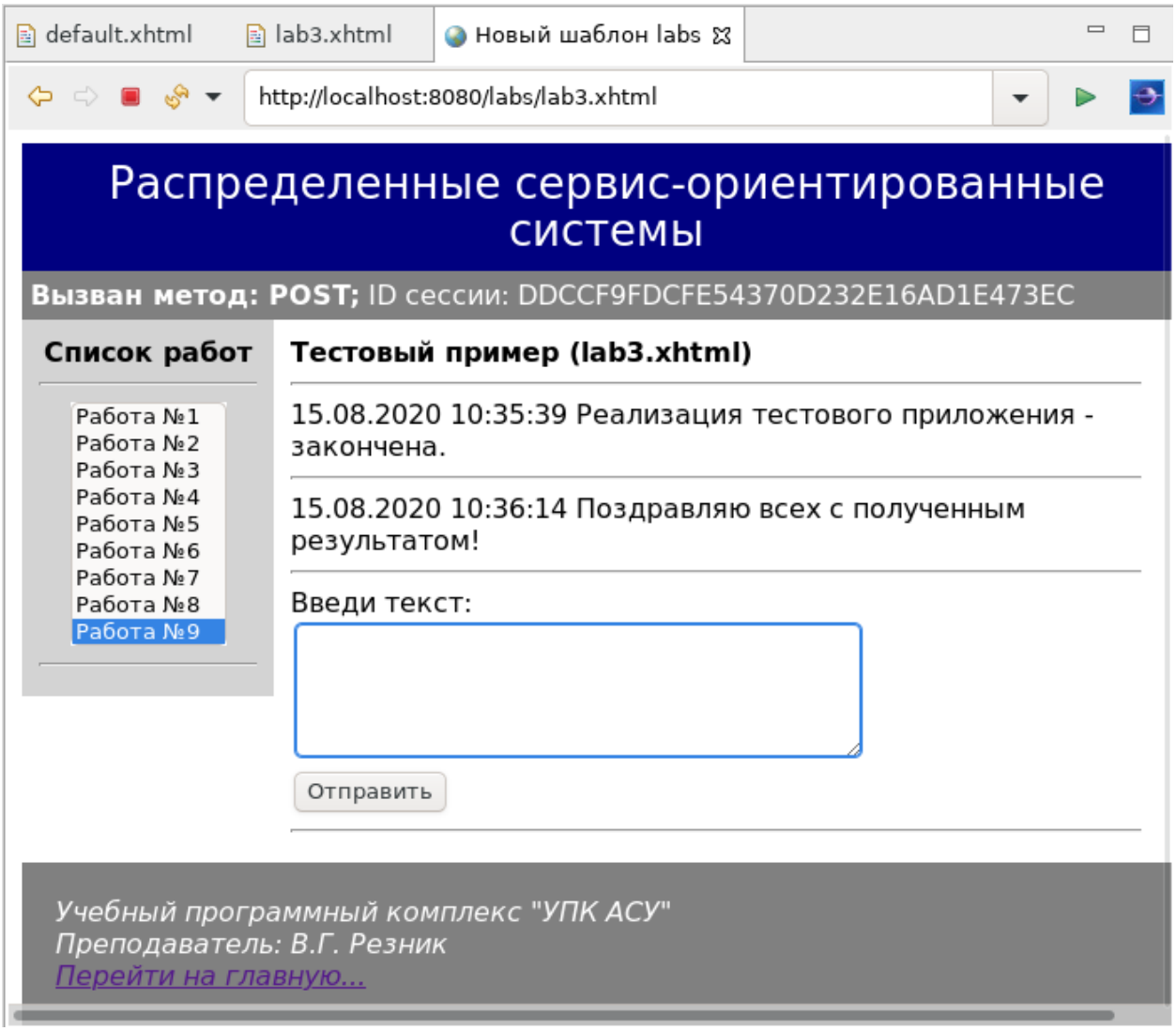

Рисунок 2.25 — Результат полной реализации тестового приложения

2.4 Реализация уровня интерфейса сервисов

Если первое назначение технологии JSF — формирование ***уровня бизнес-логики*** для архитектурной бизнес-парадигмы предприятия (см. рис. 1.12, стр. 32), то второе ее назначение — формирование ***уровня интерфеса сервисов*** этой же парадигмы.

В первой главе данного учебного пособия приведен тестовый пример, выполненный по JSP-технологии. Учебная цель этого примера — проверка и демонстрация работоспособности сервера приложений Apache TomEE и интегрированной инструментальной среды разработки Eclipse EE.

Поскольку тестовый пример — учебный, то прикладная составляющая его — минимальна. Она состоит в сохранении всех поступающих от браузера текстовых сообщений в одной приватной переменной ***msgs*** типа ***String***. Когда

браузер запрашивал содержимое этой переменной, она возвращалась в ответе как часть HTML-страницы и отображалась браузером со всеми вставленными в нее тегами: переводом строки ***
*** и линией подчеркивания ***<hr>***.

В проекции шаблона проектирования MVC, HttpServlet ***TestTomee*** выполняет функции контроллера и модели, а поскольку сервер приложений кэширует сервлеты в памяти ЭВМ, то это создает преимущество по сравнению с реализацией приложения на основе PHP-технологии, но создает общие проблемы, в плане реализации распределенных систем.

Основная проектная проблема тестового приложения — ***сильная связанность*** контроллера и модели, обусловленная самой постановкой задачи.

Сохранение в сервлете ***TestTomee*** разделяемой переменной ***msgs*** делает тестовое приложение сильносвязанной системой, имеет следующие пос-ледствия:

а) ***делает сервлет уникальным*** и пригодным для решения только одной конкретной задачи;

б) требует от сервлета ***решения проблемы многопользовательского доступа***, в виде сохранности данных и разграничения прав доступа.

Технология JSF предоставляет большие возможности по реализации уровня бизнес-логики бизнес-парадигмы предприятия, но не обеспечивает автоматического создания слабосвязанной системы.

Рассмотреная в предыдущем подразделе реализация тестового приложения средствами JSF-технологии, имела своей целью изучение средств представления уровня бизнес-логики. Это должно дать и положительные результаты в обучении. Но недостатки самой постановки задачи могут иметь негативные последствия:

а) компонент-подложка ***TestTomee*** имеет несвойственную таким задачам область действия ***@ApplicationScope***, что делает систему сильносвязанной;

б) остаются проблемы многопользовательского доступа;

в) сложные по структуре сообщения должны определяться в виде классов и соответствующим образом реализовываться.

Учебная цель данного подраздела — изучение той части JSF-технологии, которая обеспечивают реализацию интерфейса сервисов уровня предприятия.

Неудивительно, что постановки задач, в стиле рассмотренного ранее те-

стового примера, приводят к мнению, что технология JSF, да и сама программная платформа Java EE, слишком сложны для практического применения и вызывают стремление использовать другие, более «*популярные*», инструменты.

Поскольку объем возможностей JSF — достаточно большой, в следующих пунктах данного подраздела ограничимся наиболее важными примерами использования CDI-компонент, которые будем конкретизировать, опираясь на базовый шаблон ***lab3Templ.xhtml*** проекта ***labs***.

2.4.1 Жизненый цикл компонентов-подложек

> В пункте 2.2.3 «*Модель в виде компонентов-подложек*» было представлено пять вариантов области их действия (см. таблица 2.2, стр. 76).

Наиболее важными для наших рассуждений являются области действия, представленные аннотациями: @ApplicationScoped, @SessionScoped и @RequestScoped.

1. **@ApplicationScoped** — аннотация, соответствующая компоненте приложению, жизненый цикл которой начинается с запуском сервера приложений и заканчивается с остановкой сервера. В проекции на бизнес-парадигму предприятия, она ***соответствует уровню приложений***, и поэтому не должна использоваться. Исключением являются внутренние потребности проекта на уровне интерфеса сервисов, например, когда нужно защитить приложение, ограничив права доступа, или для учебных (проектных) работ, моделируя работу бизнес-приложения.
2. **@SessionScoped** — аннотация, соответствующая сессии пользователя с момента, когда браузер подключился к серверу приложений и web-контейнер создал сессию, до момента, когда web-контейнер закрыл сессию. В проекции на бизнес-парадигмы предприятия, она ***соответст-вует уровню интерфеса сервисов***, поэтому является основным объектом нашего внимания. На компоненты-подложки с таким жизненным циклом должны опираться Facelets-компоненты базовых шаблонов, когда они предоставляют доступ к расположенным на них формам, компонентам ссылок и другим активным компонентам XHTML-ресурсов. Как CDI-компоненты ***допускают инъекции объектов***, аннотированных жизенными циклами приложений или сессий.
3. **@RequestScoped** — аннотация, соответствующая непосредственному запросу браузера с момента получения его сервером приложений и до момента, когда сервер отправил браузеру ответ. Фактически все активные компоненты XHTML-ресурсов должны использовать подложки такого типа. Как CDI-компоненты ***допускают инъекции объектов***, аннотированных жизенными циклами приложений или сессий.

Facelets-шаблону изучаемой дисциплины необходима компонента-подложка с жизненным циклом сессии.

2.4.2 Компонента Users с ЖЦ @ApplicationScoped

Уровень приложений обеспечивает компонента ***Users***, доступ к которой осуществляет только компонента-подложка ***RSOS*** с жизненным циклом (ЖЦ, области действия) ***@SessionScoped***.

Уровень приложений может быть реализован различными способами, некоторые из которых будут рассмотрены в последующих главах. В данном пунке мы реализует POJO-класс, который приватно хранит несколько имен пользователей с паролями и имеет метод ***getUser(...)***, и который по заданному имени и паролю возвращает:

а) ***имя пользователя***, если такой пользователь имеется и пароль указан провильно;

б) ***пустую строку (null)***, если имя и/или пароль заданы неправильно.

Для учебных целей, хранилище имен и паролей пользователей реализовано в виде Java-коллекции ***HashMap<String,String>***, работа с которой обеспечивается следующими определениями:

а) **HashMap<String,String> map = new HashMap<String,String>()** — начальное создание пустой коллекции;

б) **map.put("имя", "пароль")** — добавление пары слов в коллекцию;

в) **map.get("имя")** — получение пароля по имени пользователя;

г) **map.remove("имя")** — удаление пары <**"имя", "пароль**"> из коллекции.

Исходный текст класса ***Users*** представлен на листинге 2.19.

Листинг 2.19 — Исходный текст класса Users.java проекта labs

```
package asu.rsos;

import java.util.HashMap;
import javax.annotation.PostConstruct;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named
@ApplicationScoped
public
```

```
class Users
{
	// Хранилище имен и паролей
	private HashMap<String,String> map;

	// Пустой конструктор
	public Users()
	{
		map = new HashMap<>();
		map.put("asu", "upkasu");
		map.put("upk", "upkasu");
	}

	// Для демонстрации
	@PostConstruct
	private void test()
		{
		System.out.println("Объект класса Users - стартовал...");
	}
	// Проверка пользователя и возврат его имени
	public String getUser (String user, String password)
	{
		System.out.println("Users:getUser() = " + user + "/" + password);
		if(user == null || password == null || user.length() == 0
			|| password.length() == 0) return null;

		String pwd = map.get(user);
		System.out.println("Users:map.get(user) = " + user + "/" + pwd);

		if(password.equals(pwd)) return user;

		return null;
    }
}
```

2.4.3 Компонента RSOS с ЖЦ @SessionScoped

RSOS — компонента-подложка с жизненным циклом сессии, реализующая интерфейс с уровнем приложения, представленного классом **Users**.

В общем случае, задача компонента RSOS — обеспечить интерфейс со всеми объектами уровня приложений.

В пределах локальной учебной задачи, компонента RSOS, являясь на сервере представителем клиента, должна обеспечить:

а) ***доступ*** к объекту класса Users и ***сохранение*** имени авторизованного пользователя;

б) ***сохранение*** номера или имени выполняемой пользователем учебной работы;

в) ***хранилищем*** «*карты*» выполняемой работы и соотвествующим ей ресурсом XHTML-файла;

г) ***по мере развития*** учебного проекта ***labs***, например, при подключении до-

полнительных объектов уровня приложений, эта компонента должна добавляться нужными данными и методами.

Учитывая достаточно прозрачное алгоритмическое содержание класса RSOS, обратимся сразу к содержанию исходного кода, представленного на листинге 2.20.

Листинг 2.20 — Исходный текст класса RSOS.java проекта labs

```
package asu.rsos;

import java.io.Serializable;
import java.util.HashMap;

import javax.annotation.PostConstruct;
import javax.enterprise.context.SessionScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@SessionScoped
public
class RSOS implements Serializable
{
	/**
	 * Стандартный идентификатор для сериализации
	 */
	private static final long serialVersionUID = 1L;

	/**
	 * Инъекция объекта класса Users
	 */
	@Inject
	private Users users;

	// Хранилище имени пользователя
	private String user = null;
	// Хранилище имени работы
	private String work = null;

	// Хранилище имен работ и соответствующих им ресурсов
	private HashMap<String,String> map;

	/**
	 * Пустой конструктор
	 */
	public RSOS()
	{
		map = new HashMap<>();
		map.put("Работа №1", null);
		map.put("Работа №2", null);
		map.put("Работа №3", "lab3");
		map.put("Работа №4", "lab4");
		map.put("Работа №5", "lab5");
		map.put("Работа №6", "lab6");
```

```
                map.put("Работа №7", "lab7");
                map.put("Работа №8", "lab8");
                map.put("Работа №9", "lab9");
        }
        // Для демонстрации
        @PostConstruct
        private void test() {
                System.out.println("Объект класса RSOS - стартовал...");
        }

        // Стандартная обработка атрибутов
        public String getUser()
        {
                //System.out.println("RSOS: getUser() = " + user);
                return user;
        }
        public void setUser(String user)
        {
                System.out.println("RSOS: setUser() = " + user);
                this.user = user;
        }

        public String getWork() {return work;}

        public void   setWork(String work) {this.work = work;}

        /**
         * Метод проведения авторизации
         */
        public void loginRSOS(String user, String password)
        {
                System.out.println("RSOS:loginRSOS() = "
                                + user + "/" + password);
                this.user = users.getUser(user, password);
        }

        /**
         * Метод отказа от авторизации
         */
        public void logoutRSOS()
        {
                user = null;
                System.out.println("RSOS:logoutRSOS() = " + user);
        }

        /**
         * Метод получения имени ресурса (имени XHTML-файла)
         */
        public String getXHTML()
        {
                return map.get(work);
        }
}
```

Обратите внимание на ряд особенностей, которые присущи реализации этого класса:

а) класс должен иметь аннотации ***@Named*** и ***@SessionScoped***, а также — поддерживать ***интерфейс сериализации***;

б) используется аннотация ***@Inject*** для подключения объекта класса ***Users***;

в) используется аннотация ***@PostConstruct*** — исключительно с целью демонстрации ее действия.

2.4.4 Компоненты-подложки с ЖЦ @RequestScoped

Для обеспечения слабой связанности разрабатываемого проекта, все компоненты подложки с жизненным циклом запроса должны взаимодействовать только с компонентами-подложками уровня сессии.

Идея состоит в том, чтобы функциональность компоненты-подложки с ЖЦ сессии развивать аддитивно, не изменяя уже разработанного интерфейса для компонент-подложек с ЖЦ запроса. В таком случае, проект будет развиваться постепенно, наполняя свою функциональность, что очень важно для больших проектов.

Задачи компонентов подложек с ЖЦ запроса — получить данные с уровня сессии, преобразовать эти данные в нужный формат и обеспечить функциональность XHTML-ресурсов приложения.

Что касается текущего состояния проекта ***labs***, то на данный момент реализованы три тестовых примера: «*Запуск тестового примера...*», «*Показать базовый шаблон*» и «*Лабораторная работа №3*». Они демонстрируют некоторые базовые возможности JSF, но не следуют напрямую современным тенденциям развития программной платформы Java EE.

Для демонстрации современных возможностей управляемых CDI-компонент разработаем функционал авторизации пользователей проекта ***labs***, ограничившись функционалом уже разработанных классов ***Users*** и ***RSOS***. Также учтем, что базовый шаблон ***lab3Templ.xhtml*** объединяет пять XHTML-ресурсов, два из которых — ***header3.xhtml*** и ***footer3.xhtml*** являются статическими и не требуют компонент-подложек.

В условиях поставленной задачи, общая схема взаимодействия запросов браузера и компонент-подложек проекта ***labs*** может быть представлена рисунком 2.26.

Приведенная схема взаимодействия компонент-подложек JSF соответствует идее слабосвязанных систем и отражает постановку трех задач:

а) ***задачи авторизации пользователя***, представленная файлом Web-ре-сурса ***auth3.xhtml*** и компонентом-подложкой ***Auth3***;

б) ***задачи отображения позаголовка***, представленная файлом Web-ре-сурса ***subheader3.xhtml*** и компонентом-подложкой ***SubHeader3***;

в) ***задачи отображения меню работ***, представленная файлом Web-ре-сурса ***menus3.xhtml*** и компонентом-подложкой ***Menus3***.

Последующие пункты демонстрируют решение этих задач.

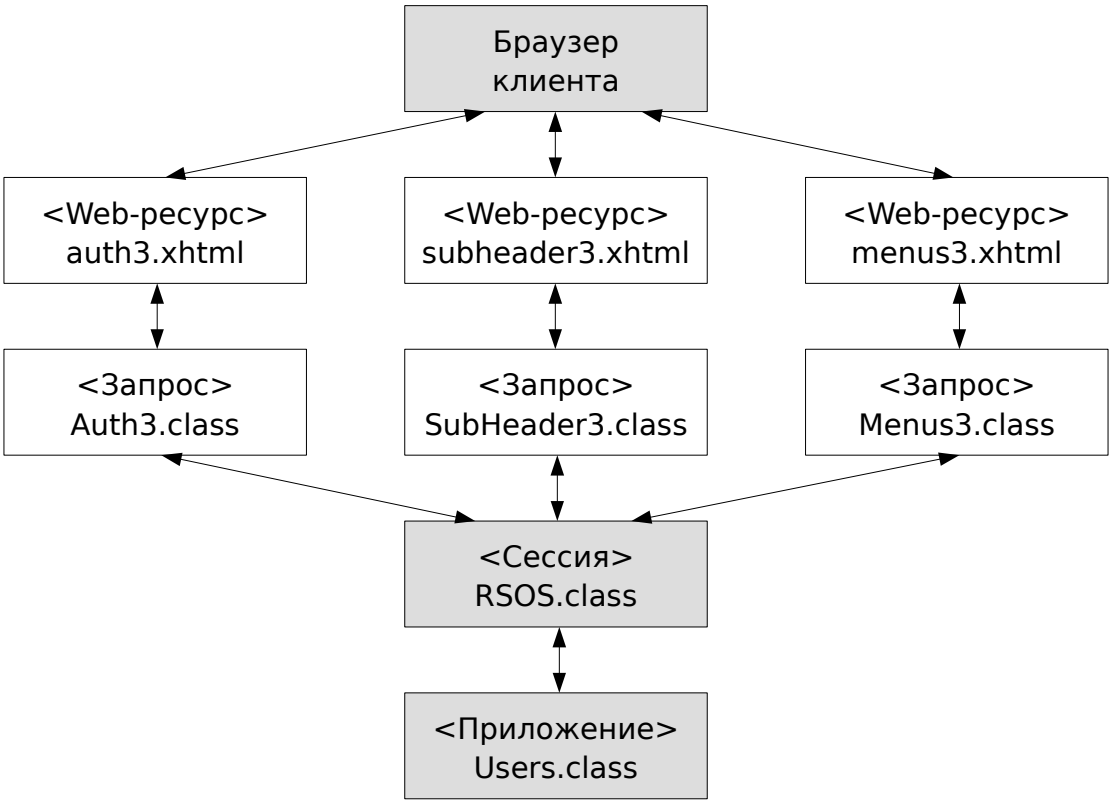

Рисунок 2.26 — Схема взаимодействия браузера и компонент-подложек JSF проекта labs

2.4.5 Приложение авторизации пользователя

Функциональность задачи авторизации пользователя обеспечивается CDI-компонентом ***Auth3*** с ЖЦ @RequestScoped.

Решение задачи авторизации обеспечивается тремя объектными приватными данными: инекцией CDI-компоненты RSOS (***rsos***) и двумя строковыми переменными (***user*** и ***password***). Ее непосредственная функциональность обеспечивается двумя методами:

а) ***String loginAuth()*** — авторизация по имени и паролю пользователя с последующим переходом на заданный Web-ресурс;

б) ***String logoutAuth()*** — сброс результата авторизации с последующим переходом на заданный Web-ресурс.

Результат реализации ***Auth3*** представлен на листинге 2.21.

Листинг 2.21 — Исходный текст класса Auth3.java проекта labs

```
package asu.rsos;

import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@RequestScoped
public
class Auth3
{
    /**
     * Инъекция объекта класса RSOS
     */
    @Inject
    private RSOS rsos;

    // Атрибут имени  пользователя
    private String user = "";
    // Атрибут пароля пользователя
    private String password = "";

    /**
     * Пустой конструктор
     */
    public Auth3() {}

    // Для демонстрации
    @PostConstruct
    private void test() {
      System.out.println("Объект класса Auth3 - стартовал...");
    }

    // Стандартная обработка атрибутов
    public String getUser()
    {
      System.out.println("Auth3: getUser() = " + user);
      return user;
    }

    public void   setUser(String user)
    {
      System.out.println("Auth3: setUser() = " + user);
      this.user = user;
    }

    public String getPassword() {return password;}

    public void   setPassword(String password) {this.password = password;}

    /**
     * Метод проведения авторизации и
```

```
     * переход на нужный ресурс
     */
    public String loginAuth()
    {
      rsos.loginRSOS(user, password);
      user = rsos.getUser();
      if(user == null || rsos.getWork() == null)
            return "default";
      // Запускаем ресурс работы
      String ss = rsos.getXHTML();
      if(ss == null) return "default";
      return ss;
    }
    /**
     * Метод сброса результата авторизации и
     * переход на нужный ресурс
     */
    public String logoutAuth()
    {
      rsos.logoutRSOS();
      user = null;
            return "default";
    }
}
```

За отображение задачи авторизации отвечает Web-ресурс ***auth3.xhtml***, переопределяющий шаблон ***lab3Templ.xhtml*** с именем "***context***".

Заявленный Web-ресурс полностью опирается на функциональность своей компоненты-подложки и содержит: поля ввода имени и пароля пользователя, а также кнопку запроса, представленные на листинге 2.22. Результат разработки этого приложения представлен на рисунке 2.27.

Листинг 2.22 — Исходный текст файла auth3.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Приложение auth3</title>
</head>

<body>
      <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">
      <ui:define name="context">
      <div style="color:black; width:100%; padding:10px">
```

```
                    <b>Авторизация пользователя</b> <hr/>
        <h:form>
              <h:panelGrid columns="2">
                  <h:outputText  value="Имя:"></h:outputText>
                  <h:inputText   value="#{auth3.user}"/>
                  <h:outputText  value="Пароль:"></h:outputText>
                  <h:inputSecret value="#{auth3.password}"></h:inputSecret>
              </h:panelGrid>
              <h:commandButton value="Авторизация"
                    action="#{auth3.loginAuth()}"></h:commandButton>
              <h:commandButton value="Отказаться"
                    action="#{auth3.loginAuth()}"></h:commandButton>

        </h:form>
        </div>
        </ui:define>
        </ui:composition>
</body>
</html>
```

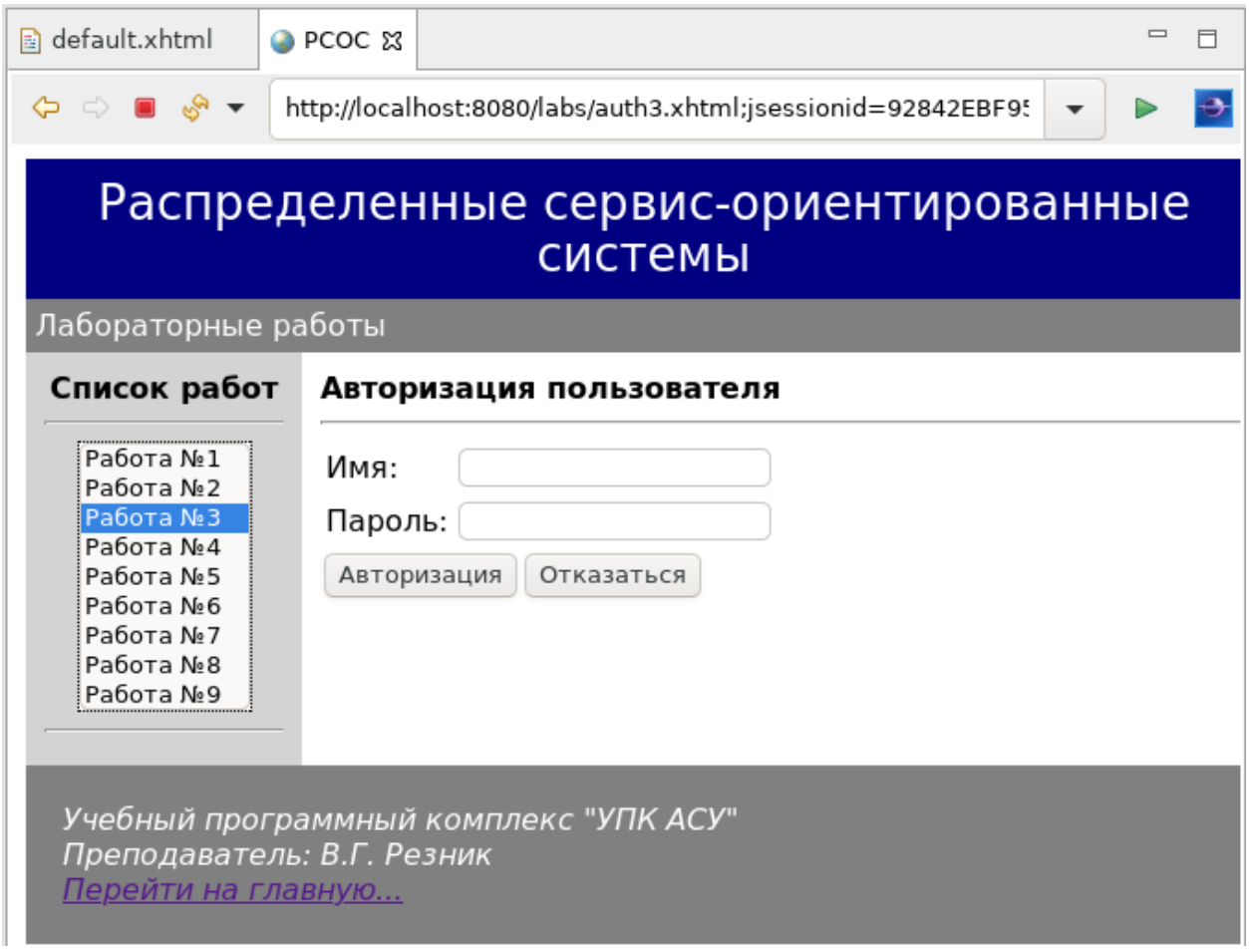

Рисунок 2.27 — Результат разработки приложения авторизации пользователя

2.4.6 Компонента подзаголовка проекта

Функциональность отображения подзаголовка шаблона проекта ***labs*** обеспечивается новой CDI-компонентой ***SubHeader31*** с ЖЦ ***@RequestScoped***.

Идея нового подзаголовка проекта ***labs*** предполагает, что он должен отображать название лабораторной работы, имя пользователя и кнопку авторизации, предоставляемые данными сессионной компоненты ***RSOS***.

Подложка подзаголовка обеспечивает инъекцию CDI-компоненты ***rsos***, а также — сохранение двух строковых переменных: ***user*** и ***password***. Их функциональность обеспечивают четыре метода: ***getUser()***, ***getWork()***, ***getButton()*** и ***setLogin()***, а исходный текст этой подложки представлен на листинге 2.23.

Листинг 2.23 — Исходный текст класса SubHeader31.java проекта labs

```
package asu.rsos;
import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@RequestScoped
public
class SubHeader31 {
    /**
     * Инъекция обекта класса RSOS
     */
    @Inject
    private RSOS rsos;

    // Имя пользователя
    private String user = null;
    // Наименование работы
    private String work = null;

    /**
     * Пустой конструктор
     */
    public SubHeader31() {}

    // Для демонстрации
    @PostConstruct
    private void test() {
      System.out.println("Объект класса SubHeader31 - стартовал...");
    }

    // Стандартная обработка атрибутов
    public String getUser()
    {
      user = rsos.getUser();
```

```
      if(user == null) return "Не выбран";
      return user;
    }
    public String getWork()
    {
      work = rsos.getWork();
      if(work == null) return "Не выбрана";
      return work;
    }

    public String getButton()
    {
      user = rsos.getUser();
      if(user == null) return "Войти";
      return "Выйти";
    }

    /**
     * Метод проведения авторизации
     */
    public String setLogin()
    {
      user = rsos.getUser();
      if(user == null) return "auth3";
      rsos.setUser(null);
      return "default";
    }
}
```

За отображение подзаголовка отвечает Web-ресурс ***subheader31.xhtml***, определяющий загрузку нового Web-компонента шаблона ***lab3Templ.xhtml*** с именем "***subheader***".

Заявленный Web-ресурс полностью опирается на функциональность своей компоненты-подложки и содержит: поля ввода имени и пароля пользователя, а также кнопку запроса, представленные на листинге 2.24.

Листинг 2.24 — Исходный текст файла subheader31.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
            <title>Вид занятий</title>
</head>

<body>
            <div  style="background-color:gray;width:100%;
```

```
                        color:white;padding-left:20px;padding-right:20px;">

        <h:form id="subForm">

              <table align="center" width="100%" border="0"
                    cellpadding="0" cellspacing="0">
              <tr>
                    <td><h:outputLabel value=" Лабораторные работы: " />
                    </td>
                    <td><i><h:outputLabel value="#{subHeader31.work}" /></i>
                    </td>
                    <td><i><h:outputLabel value="#{subHeader31.user}" /></i>
                    </td>
                    <td><h:commandButton value="#{subHeader31.button}"
                                         action="#{subHeader31.setLogin}" />
                    </td>
              </tr>
              </table>
        </h:form>
        </div>
</body>
</html>
```

После завершения всех изменений и запуска проекта ***labs***, мы получим результат, показанный на рисунке 2.28.

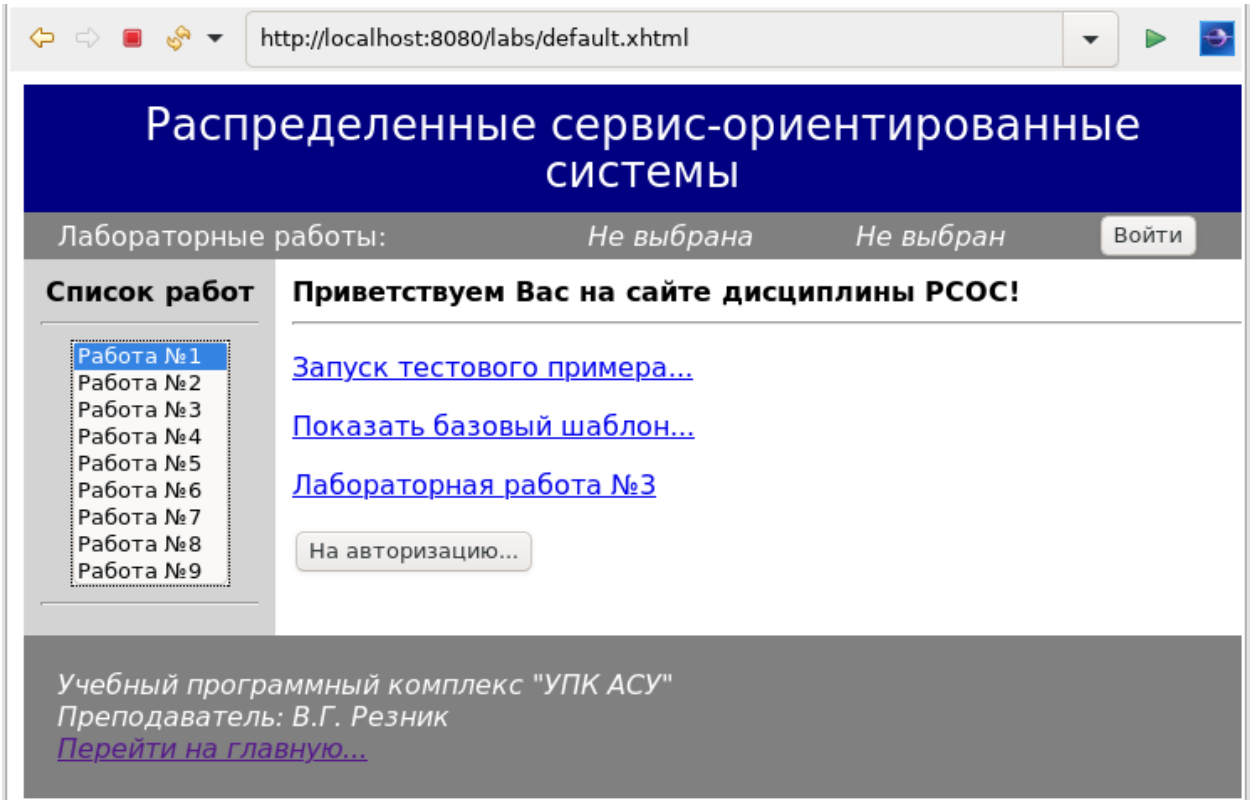

Рисунок 2.28 — Результат запуска labs без авторизации пользователя

После проведения авторизации пользователем ***upk***, результат изменится, как показано на рисунке 2.29.

Обратите внимание, что на обоих рисунках имя лабораторной работы обозначено как «*Не выбрана*». Это связано с тем, что боковая панель еще не функционирует, поэтому имя работы еще не сохраняется в сессионной компоненте ***RSOS***.

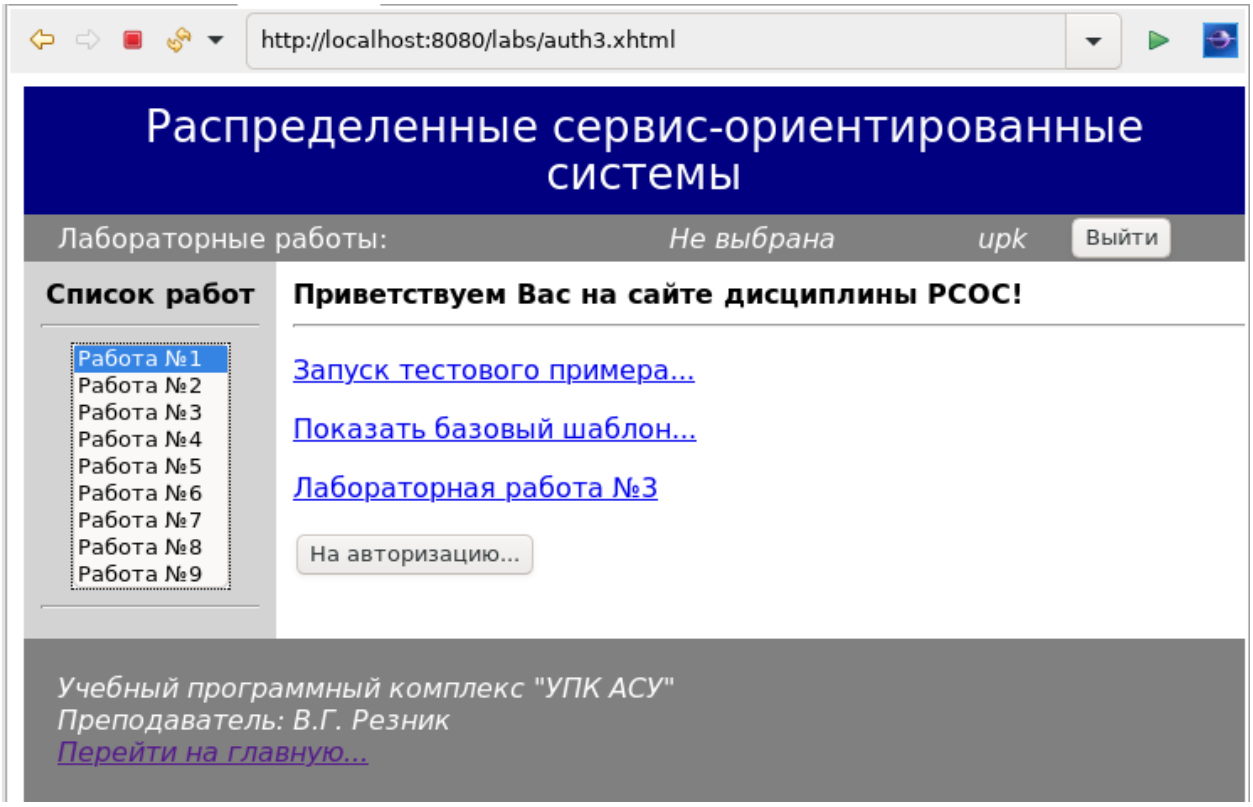

Рисунок 2.29 — Результат запуска labs после авторизации пользователя

2.4.7 Компонента меню лабораторных работ

Функциональность первого варианта задачи отображения меню лабораторных работ проекта ***labs*** обеспечивается CDI-компонентом ***Menus31*** с ЖЦ ***@RequestScoped***.

Идея боковой панели состоит в реализации функциональности проекта ***labs***, которая бы обеспечивала выбор и запуск конкретной лабораторной работы.

Наиболее простая реализация обеспечивается размещением на панели кнопок, каждая из которых запускает приложение отдельной работы. Тогда функциональность CDI-компоненты ***Menus31*** сведется к вызову только одного метода, например, метода ***run(...)***, который сохранит имя работы в сессионной компоненте типа ***RSOS***, а затем запустит нужный XHTML-ресурс, соответствующий выбранной работе.

На листинге 2.25 представлен исходный текст компоненты Menus31.

Листинг 2.25 — Исходный текст класса Menus31.java проекта labs

```
package asu.rsos;

import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@RequestScoped
public
class Menus31 {
    /**
     * Инъекция обекта класса RSOS
     */
    @Inject
    private RSOS rsos;

    // Имя пользователя
    private String user = null;
    // Наименование работы
    private String work = null;

    /**
     * Пустой конструктор
     */
    public Menus31() {}

    // Для демонстрации
    @PostConstruct
    private void test() {
      System.out.println("Объект класса Menus31 - стартовал...");
    }

    // Стандартная обработка атрибутов
    public String getUser()
    {
      user = rsos.getUser();
      if(user == null) return "Не выбран";
      return user;
    }
    public String getWork()
    {
      work = rsos.getWork();
      if(work == null) return "Не выбрана";
      return work;
    }
    public boolean isShow()
    {
      user = rsos.getUser();
      if(user == null) return false;
      return true;
    }
```

```
    /**
     * Метод запуска работ
     */
    public String run(String msg)
    {
      System.out.println("Menus31:run() = " + msg);
      rsos.setWork("Работа №" + msg);
      String ss = rsos.getXHTML();
      if(ss == null) return "default";
      return ss;
    }
}
```

Обратите внимание на метод ***isShow()***, который возвращает булевое значение ***true/false***. Благодаря ему, панель будет скрыта до тех пор, пока пользователь не авторизуется, и запуск работ не будет доступен.

> За отображение боковой панели отвечает Web-ресурс ***menus31.xhtml***. Он переопределяет Web-компоненту шаблона ***lab3Templ.xhtml*** с именем "***menus***".

Боковая панель представляет собой форму ***<h:form>***, внутри которой кнопки объеденены панелью ***<h:panelGrid>*** с параметром ***rendered***, отвечающим за показ или сокрытие всего набора кнопок. Исходный текст этого Web-ресурса приведен на листинге 2.26.

Листинг 2.26 — Исходный текст файла menus31.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:f="http://xmlns.jcp.org/jsf/core">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Боковая страница</title>
</head>
<body>

<h:form>
      <h:panelGrid rendered="#{menus31.show}" colums="1"
            style="background-color:lightgray;width:150px;padding:10px">

      <f:facet name="header">
      <h:outputText value="Список работ"/>
      </f:facet>
      <hr/>
      <h:commandButton value="Работа №1" action="#{menus31.run('1')}"/>
      <h:commandButton value="Работа №2" action="#{menus31.run('2')}"/>
      <h:commandButton value="Работа №3" action="#{menus31.run('3')}"/>
```

```
        <h:commandButton value="Работа №4" action="#{menus31.run('4')}"/>
        <h:commandButton value="Работа №5" action="#{menus31.run('5')}"/>
        <h:commandButton value="Работа №6" action="#{menus31.run('6')}"/>
        <h:commandButton value="Работа №7" action="#{menus31.run('7')}"/>
        <h:commandButton value="Работа №8" action="#{menus31.run('8')}"/>
        <h:commandButton value="Работа №9" action="#{menus31.run('9')}"/>
        <hr/>
        </h:panelGrid>
</h:form>

</body>
</html>
```

Запуск проекта до авторизации пользователя — на рисунке 2.30.

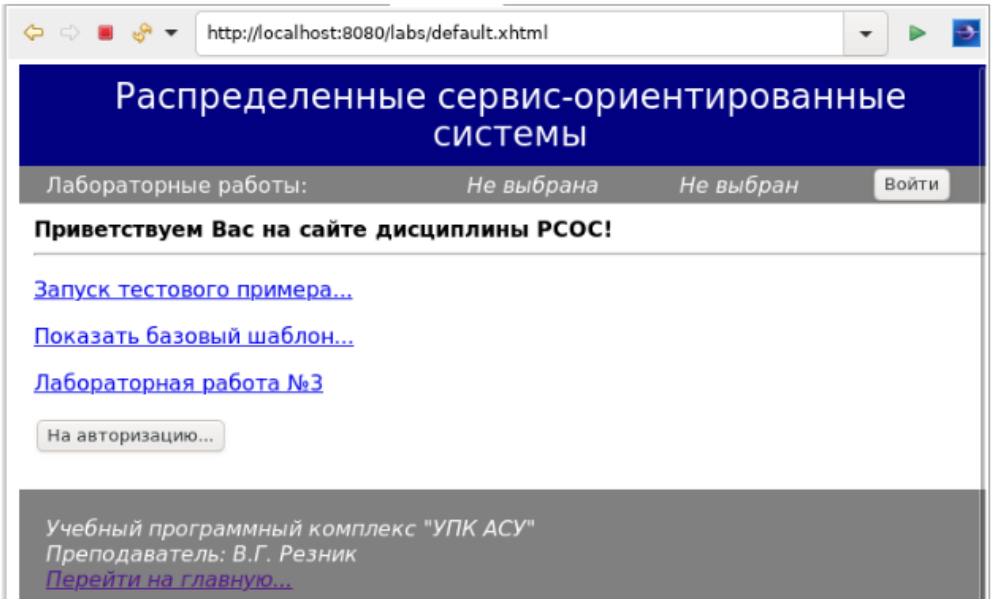

Рисунок 2.30 — Запуск labs с боковой панелью до авторизации пользователя

Запуск ***labs*** после авторизации пользователя показан на рисунке 2.31.

2.4.8 Второй вариант меню лабораторных работ

Отображение боковой панели может быть представлено в другой форме, например, в виде списка лабораторных работ и одной кнопки, формирующей запрос к серверу. Такой подход может оказаться более эстетически обоснованным, хотя реализуется дополнительными инструментами технологии JSF.

Рассмотрим новый вариант боковой панели, показанный на листинге 2.27.

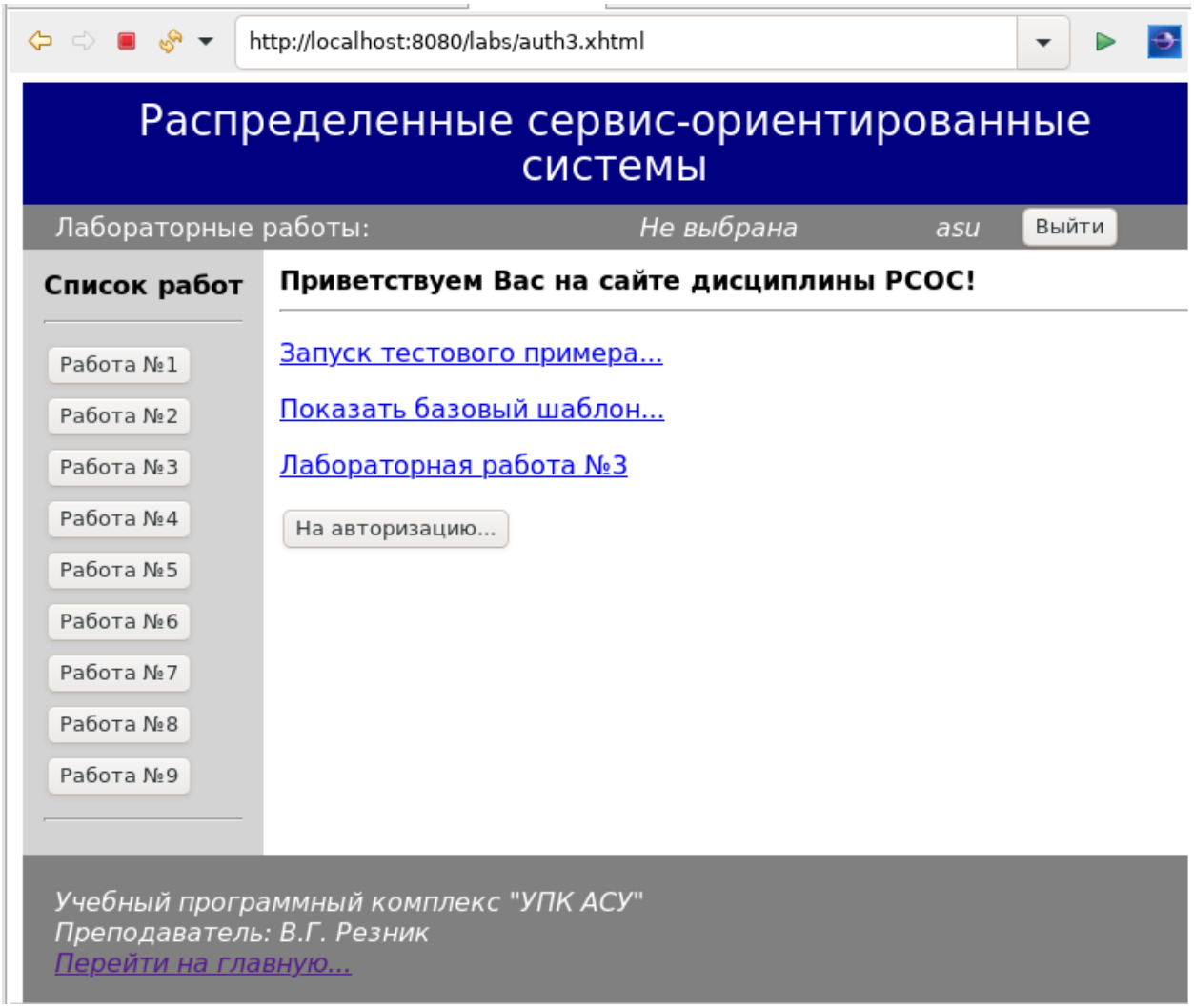

Рисунок 2.31 — Запуск labs после авторизации пользователя

Листинг 2.27 — Исходный текст файла menus32.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:h="http://xmlns.jcp.org/jsf/html"
            xmlns:f="http://xmlns.jcp.org/jsf/core">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
            <title>Боковая страница</title>
</head>

<body>
<h:form>
      <h:panelGrid rendered="#{menus32.show}" columns="1"
            style="background-color:lightgray;width:150px;padding:10px">
```

```
      <f:facet name="header">
      <h:outputText value="Список работ"/>
      </f:facet>
      <hr/>

      <h:selectOneListbox id="list" align="center"
                  value="#{menus32.work}"
                  valueChangeListener="#{menus32.valueChanged}"
                  onchange="submit()">

      <f:selectItems value="#{menus32.data}"/>

      </h:selectOneListbox>
      <hr/>
      <h:commandButton action="#{menus32.run}" value="Запустить" />
      </h:panelGrid>
</h:form>
</body>
</html>
```

Для реализации такого представления необходима новая компонента-подложка, например, ***Menus32***. Кроме того, новой подложке требуется дополнительная функциональная поддержка и представление данных:

а) тег <***h:selectOneListbox***> требует реализации метода ***valueChanget()***, обслуживающего выбор пунктов меню;

б) тег <***f:selectItems***> требует доступа к списку объектов, каждый из которых представляет парные строки.

Для реализации функциональности подложки ***Menus32*** создадим дополнительный CDI/POJO-класс ***DString***, исходный текст которого представлен на листинге 2.28.

Листинг 2.28 — Исходный текст класса DString.java проекта labs

```
package asu.rsos;

import javax.inject.Named;

@Named
public class DString {
    /**
     *  Пара строк для тега <f:selectItems>
     */
    private String name;
    private String desc;

    public DString() {}

    public DString(String name, String desc) {
        this.name = name;
        this.desc = desc;
    }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDesc() {
        return desc;
    }
    public void setDesc(String desc) {
        this.desc = desc;
    }
    @Override
    public String toString() {
        return this.name;
    }
}
```

Теперь компонента-подложка ***Menus32***, предназначенная для новой боковой панели может быть представлена листингом 2.29, а результат запуска проекта ***labs*** с новой боковой панелью показан на рисунке 2.32.

Листинг 2.29 — Исходный текст класса Menus32.java проекта labs

```
package asu.rsos;

import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.faces.event.ValueChangeEvent;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@RequestScoped
public
class Menus32 {
    /**
     * Инъекция обекта класса RSOS
     */
    @Inject
    private RSOS rsos;

    // Имя пользователя
    private String user = null;
    // Наименование работы
    private String work = null;
    // Содержимое <h:selectOneListbox>
    private List<DString> data;
    /**
     * Пустой конструктор
     */
    public Menus32()
```

```
{
  data = new ArrayList<>();
  data.add(new DString("Работа №1", "1"));
  data.add(new DString("Работа №2", "2"));
  data.add(new DString("Работа №3", "3"));
  data.add(new DString("Работа №4", "4"));
  data.add(new DString("Работа №5", "5"));
  data.add(new DString("Работа №6", "6"));
  data.add(new DString("Работа №7", "7"));
  data.add(new DString("Работа №8", "8"));
  data.add(new DString("Работа №9", "9"));
}

// Для демонстрации
@PostConstruct
private void test() {
  System.out.println("Объект класса Menus32 - стартовал...");
}

// Стандартная обработка атрибутов
public String getUser()
{
  user = rsos.getUser();
  if(user == null) return "Не выбран";
  return user;
}
public String getWork()
{
  work = rsos.getWork();
  if(work == null) return "Работа №1";
  return work;
}
public void setWork(String work) {this.work = work;}

public List<DString> getData()
{
  return data;
}

// Показывать или не показывать компоненту
public boolean isShow()
{
  user = rsos.getUser();
  if(user == null) return false;
  return true;
}

/**
 * Метод запуска работ
 */
public String run()
{
  String ss = rsos.getXHTML();
  if(ss == null) ss = "default";
  System.out.println("Menus32:run() = " + ss);
  return ss;
}
```

```
    // Обработчик события
    public void valueChanged(ValueChangeEvent e)
    {
      work = (String) e.getNewValue();
      System.out.println("Menus32:valueChanged() = "
                  + work);
      rsos.setWork(work);
    }
}
```

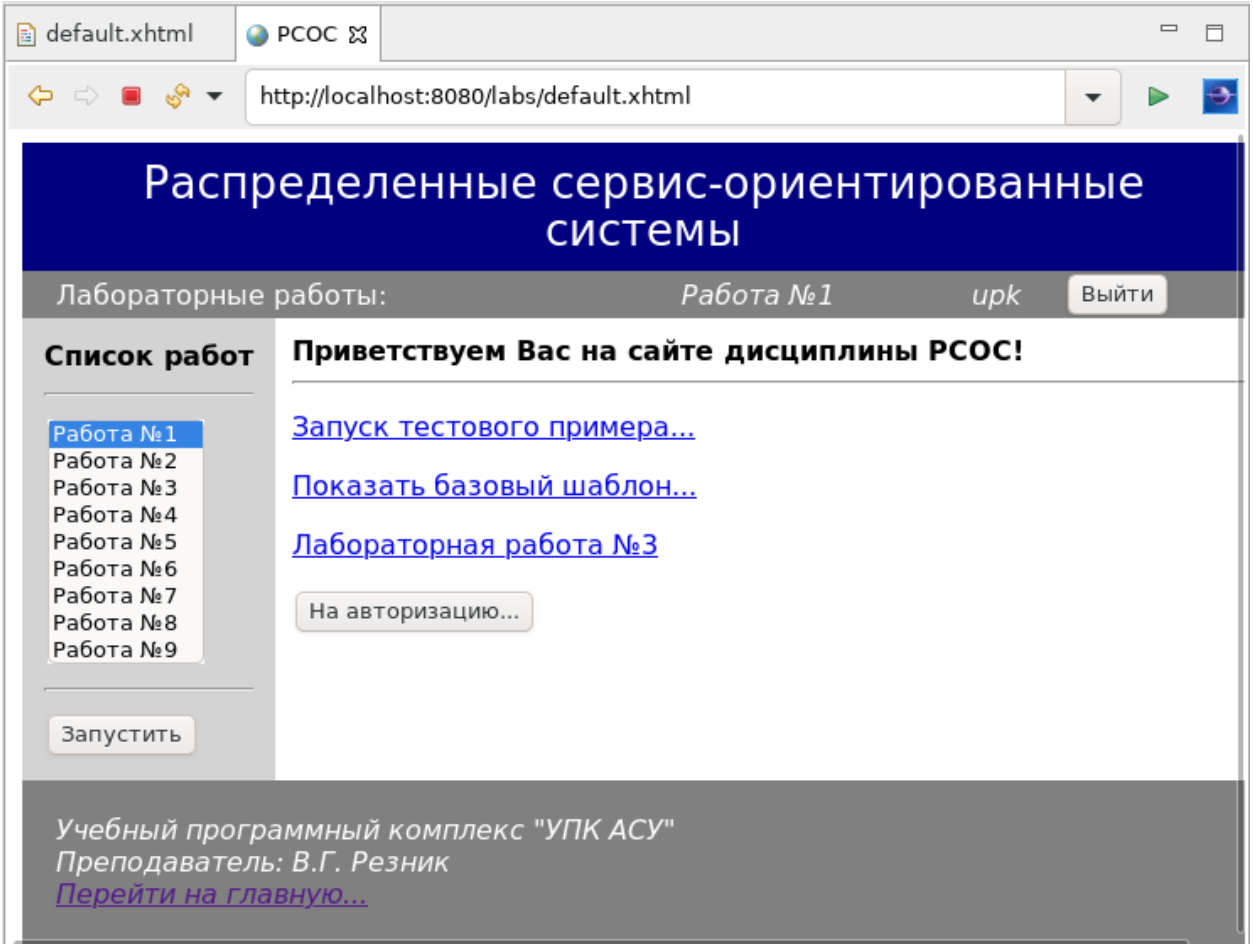

Рисунок 2.32 — Запуск labs после авторизации пользователя

Вопросы для самопроверки

1. Что означает понятие: «*Тонкий клиент*»?
2. Для чего используется десткриптор развертывания web.xml?
3. Что означает понятие: технология AJAX?
4. В чем состоят различия и сходства понятий PHP и HttpServlet?
5. Что такое — трехзвенная архитектура приложения?
6. Чем отличается технология AJAX от технологии JavaScript?
7. Как представить компоненту JSF шаблоном проектирования MVC?
8. Что такое — компонент-подложка?
9. Какие области действия бывают у компонент-подложек?
10. Что такое — компонента CDI и какая аннотация для нее является обязательной?
11. Изобразите схему жизненного цикла компонента подложки?
12. Что обозначают аннотации @PostConstruct и @PreDestroy?
13. Что такое — Facelets-компоненты?
14. В чем состоит отличие и сходство технологий JSP и JSF?
15. Что такое — шаблон JSF-проекта?
16. Сколько этапов JSF выделяет для жизненного цикла обработки запроса?
17. Что обозначает FacesContext?
18. Изобразите трехзвенную архитектуру взаимодействия «*Клиент-сервер*»?
19. Что такое — Expression Language и где он используется?
20. Для чего в среде разработки Eclipse EE предназначен каталог WEB-INF?

3 Тема 3. Современные способы доступа к данным

Согласно бизнес-парадигме архитектуры предприятия (см. пункт 1.3.4 главы 1, рисунок 1.12, стр. 39) физическая инкапсуляция сервисов размещается на третьем (нижнем) уровне — ***уровне приложений***.

Бизнес-приложения всегда ассоциируются с бизнес-моделями, бизнес-данными и хранилищами этой информации.

Классический подход построения бизнес-моделей и обработки данных связан с использованием реляционных СУБД, в которых выделяются табличные сущности, связанные различными отношениями и требованиями нормальных форм. Доступ к этим моделям и данным осуществляется на языке ***SQL*** (S*tructured Query Language*), который для каждой СУБД поддерживается специальным программным обеспечением — ***драйверами*** СУБД.

Проблема классического подхода доступа к данным — различие модельных представлений обработчиков данных, использующих объектный подход и модельных представлений СУБД, использующих язык SQL. В результате, разработчики бизнес-приложений вынуждены постоянно использовать два языка: язык для доступа к данным и язык для их обработки.

Современный подход доступа к бизнес-информации, который настоятельно рекомендуется программной платформой Java EE, — объектно-реляционное отображение (***ORM, Object-Relational Mapping***), заключающееся в объединении миров баз данных и объектов.

Учебная тема данной главы — краткое изучение технологии ***JPA*** (*Java Persistence API*), являющейся наиболее предпочтительной технологией предлагаемой платформой Java EE.

Учебный материал данной главы излагается с учетом того, что студент уже изучил технологию JDBC применительно к СУБД Apache Derby, в бакалаврском курсе «*Распределенные вычислительные системы*». Этот факт должен помочь студенту лучше разобраться в различиях классического и современного подходов.

В целом, платформа Java EE предлагает несколько других фрейворков, реализующих ORM: ***Hibernate***, ***TopLink*** и ***Java Data Objects*** (*JDO*). Среди них, технология ***JPA*** считается наиболее перспективной, поэтому и рассмотрена в данном учебном пособии, а, что касается самих этапов изучения технологии JPA, то они представлены в трех частях:

а) ***подраздел 3.1*** — описывает конкретную прикладную задачу;

б) ***подраздел 3.2*** — предоставляет необходимый теоретический материал;

в) ***подраздел 3.3*** — предоставляет вариант реализации этой задачи.

3.1 Учебная инфраструктура темы

Современные технологии доступа к данным основаны на общей парадигме слабосвязанных систем и максимальном управлении взаимодействием с СУБД с помощью сервиса контейнеров платформы Java EE.

В терминах данной дисциплины, бизнес-приложение — это EJB/EJB-Light компонент, управляемых собственным контейнером. Поэтому технологию JPA проще демонстрировать на конкретном простом компоненте EJB-Light, что и делается в данном подразделе.

Соотвественно, рассматриваемая компонента должна быть помещена в в контейнер сервера приложений Apache TomEE, а в самом контейнере должна быть указана СУБД, обслуживаемая контейнером.

Простейшая задача, которую можно интерпретировать различными прикладными вариациями, — хранение в базе данных СУБД списка текстовых сообщений пользователей, которые можно добавлять, модифицировать и удалять. Доступ к этому списку сообщений осуществляется через Web-контейнер сервера приложений.

Конкретизируем поставленную задачу в следующих двух пунктах.

3.1.1 Учебная задача Letters (Письма)

Для последущего проектирования и реализации, учебная задача конкретизируется следующим образом.

На сервере приложений учебная задача обслуживается EJB-компонентом с именем ***Letters***, который манипулирует объектами класса ***Letter.class*** и предоставляет сессионным компонентам пользователей следующие функции:

а) ***getList()*** — получить список всех объектов класса ***Letter***, хранящихся в базе данных;

б) ***getLetter(int id)*** — получает объект класса ***Letter*** по значению его идентификатора ***id***;

в) ***addLetter(Letter letter)*** — добавляет объект класса ***Letter*** в базу данных;

г) ***deleteLetter(int id)*** — удаляет объект из базы данных;

д) ***modLetter(Letter letter)*** — модифицирует объект в базе данных.

Объект манипулирования — класс ***Letter*** имеет слудующие атрибуты (поля данных):

а) ***id*** — уникальный идентификатор объекта;

б) ***date*** — дата и время создания или последней модификации;

в) ***user*** — имя авторизованного пользователя, создавшего или модифицировавшего запись;

г) ***text*** — текстовое сообщение объемом не более ***4 КБайт***.

Имеющейся информации вполне достаточно для демонстрации учебного материала изучаемой темы.

Будет большой ошибкой наделять компоненту ***Letters*** множеством разнообразных функций. Это неминуемо приведет к ее усложнению.

Правильный подход — множить функциональность на уровне сессий пользователей, а лучше — на уровне запросов к серверу приложений.

3.1.2 Корпоратиные EJB-компоненты

EJB-компоненты — это серверные компоненты, которые инкапсулируют в себе бизнес-приложения, обеспечивают взаимодействие с базами данных, заботятся о транзакциях и безопасности.

С прикладной точки зрения компоненты бывают разные. В данной главе нас интересуют EJB-компоненты, взаимодействующие с СУБД и проводящие манипуляции по сохранению данных.

Обычно, когда клиент (приложение клиента) соединяется с сервером, то сервер организует сессию и обслуживает клиента в пределах этой сессии. Если не указано другое, то в дальнейшем мы подразумеваем взаимодействие в пределах сессии.

С позиции разработчика приложений выделяется всего три типа EJB-компонент: без сохранения состояния, с сохранением состояния и одиночные.

@Stateless-компонента — EJB-компонента без сохранения состояния. Она применяется, когда решение задачи можно осуществить одним вызовом метода.

@Statefull-компонента — EJB-компонента с сохранением состояния. Она поддерживает диалоговое состояние, которое может сохраняться между вызовами методов в пределах одной сессии, поэтому полезна для решения задач, с которыми нужно справляться за несколько этапов (обращений к серверу).

@Singleton-компонента — одиночная EJB-компонента. Она совместно используется всеми клиентами и поддерживает конкурентный доступ. Контейнер сам заботится о том, чтобы для всего приложения имелся только один экземпляр.

EJB-компонента — это обычный JAVA-класс, удовлетворяющий следующим ограничениям:

1) класс должен быть снабжен одной из указанных выше аннотаций или XML-эквивалентом в дескрипторе развертывания;
2) должен реализовывать методы своих интерфейсов, если они имеются;
3) класс должен быть определен как ***public*** и не может быть определен как ***final*** или ***abstract***;
4) класс должен иметь конструктор ***public*** без аргументов, который кон-тейнер будет использовать для создания экземпляров;
5) класс не должен определять метод ***finalize()***;
6) имена методов класса не должны начинаться с ***ejb*** и они не могут быть ***final*** или ***static***;
7) аргумент и возвращаемое значение удаленного метода должны относиться к допустимым типам RMI.

Бизнес-интерфейс — это стандартный Java-интерфейс, который не расширяет никаких EJB-специфичных интерфейсов.

Распределенные системы — это мир интерфейсов, что наглядно демонстрирует рисунок 3.1.

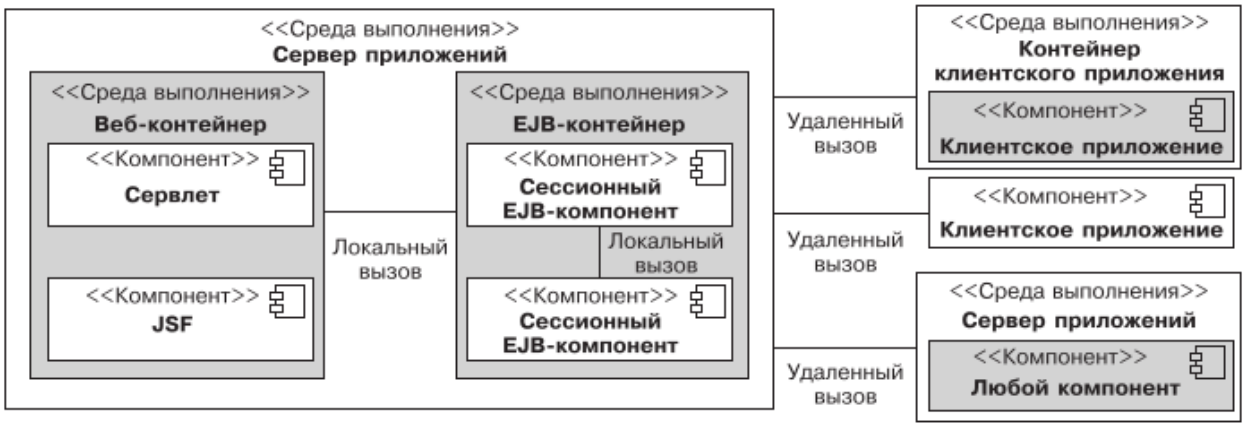

Рисунок 3.1 — Сессионные EJB-компоненты для клиентов разных типов [17]

Сессионный EJB-компонент — это управляемый объект сервера приложений, обслуживающий сессию клиента (приложения клиента) посредством локальных или удаленных вызовов. Сама возможность обслуживания клиентов определяется интерфейсами, которые реализуют и предоставляют EJB-компоненты.

В целом, интерфейсы подразделяются на локальные и удаленные, что обозначается соответствующими аннотациями:

а) ***@Local*** — обозначает локальный бизнес-интерфейс. Параметры методов *передаются по ссылке* от клиента к EJB-компоненту;

б) ***@Remote*** — обозначает удаленный бизнес-интерфейс. Параметры методов *передаются по значению* и нуждаются в том, чтобы быть сериализуемыми как часть протокола RMI.

Проведем демонстрацию сказанного формальным описанием учебной задачи ***Letters***, предполагая, что все указанные представления проводятся в учебном проекте ***lab4*** типа ***Dynamic Web Project***.

Сначала необходимо дать описание класса ***Letter***, которое реализуем в виде сериализованного POJO-класса, представленного на листинге 3.1.

Листинг 3.1 — Исходный текст класса Letter.java проекта lab4

```
package rsos.lab4;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    private int id;
    private Date date;
    private String user;
    private String text;

    // Конструкторы
    public Letter() {
    }

    public Letter(Date date, String user, String text) {
        this.date = date;
        this.user = user;
        this.text = text;
    }

    // Геттеры и сеттеры POJO-класса
    public int getId() { return id; }

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

    public Date getDate() { return date; }


    public void setDate(Date date) {
        this.date = date;
    }
```

```
    public String getUser() { return user; }

    public void setUser(String user) {
        this.user = user;
    }

    public String getText() { return text; }

    public void setText(String text) {
        this.text = text;
    }

    // Дополнительные функции форматирования
    public String toString() {
           return Integer.toString(id) + " " + date.toString()
                       + " " + user + " " + text;
    }

    public String getDateString() {
         SimpleDateFormat sdf =
                 new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
         return sdf.format(date);
    }
}
```

Теперь, на основе методов класса ***Letters***, создим два интерфейса, например, ***LocalLetter*** и ***RemoteLetter***, представленные на листингах 3.2 и 3.3.

Листинг 3.2 — Исходный текст интерфейса LocalLetter.java проекта lab4

```
package rsos.lab4;

import java.util.List;
import javax.ejb.Local;

@Local
public interface LocalLetter
{
            // Получить список писем
            List<Letter> getList();

            // Получить письмо по идентификатору
            Letter getLetter(int id);

            // Добавить письмо
            void addLetter(Letter letter);

            // Удалить письмо по идентификатору
            void deleteLetter(int id);

            // Модифицировать письмо
            void modLetter(Letter letter);
}
```

Локальный интерфейс объявляет все пять заявленных ранее методов, что обычно справедливо для локальных взаимодействий.

Листинг 3.3 — Исходный текст интерфейса RemoteLetter.java проекта lab4

```
package rsos.lab4;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface RemoteLetter
{
            // Получить список писем
            List<Letter> getList();

            // Получить письмо по идентификатору
            Letter getLetter(int id);

            // Добавить письмо
            void addLetter(Letter letter);

            // Удалить письмо по идентификатору
            void deleteLetter(int id);

            // Модифицировать письмо
            void modLetter(Letter letter);
}
```

Представленный на листинге 3.3 удаленный интерфейс с точностью до аннотации повторяет локальный интерфейс. В общем случае — это не так, потому что обычно удаленный интерфейс запрещает какие-либо методы или называет их по другому. Возможно добавляются дополнительные методы, например, обеспечивающие авторизацию доступа и другие.

EJB-компонент обеспечивает интерфейс тогда, когда он его реализует. Так, на листинге 3.4 представлена заготовка ***Letters*** «*без сохранения состояния*», обеспечивающая представление интерфейсов ***LocalLetter*** и ***RemoteLetter***.

Листинг 3.4 — Заготовка исходного текста класса Letters.java проекта lab4

```
package rsos.lab4;

import java.util.Date;
import java.util.List;
import javax.ejb.Stateless;

@Stateless
public class Letters implements LocalLetter, RemoteLetter
{
      public List<Letter> getList() {return null;}
```

```
	public Letter getLetter(int id) {
		Letter letter = new Letter(new Date(), "upk", "Сообщение");
		letter.setId(id);
		return letter;
	}
	public void addLetter(Letter letter) {}

	public void deleteLetter(int id) {}

	public void modLetter(Letter letter) {}
}
```

EJB-компонент без обеспечения интерфейса, даже если он реализует интерфейс, должен помечаться аннотацией ***@LocalBean***.

В общем случае не обязательно помечать интерфейсы аннотациями. Это можно сделать при описании EJB-компонента. Например, можно потребовать, чтобы класс ***Letters*** не обеспечивал интерфейсы, тогда начало листинга 3.4 можно заменить на содержание, показанное на листинге 3.5.

Листинг 3.5 — Измененное начало класса Letters.java проекта lab4

```
package rsos.lab4;

import java.util.List;
import javax.ejb.Local;
import javax.ejb.LocalBean;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Local(LocalLetter.class)
@Remote(RemoteLetter.class)
@LocalBean
public class Letters implements LocalLetter, RemoteLetter
{     ...     }
```

EJB-компоненты «*без сохранения состояния*» также могут вызываться удаленно как Web-службы SOAP, используя аннотацию ***@WebService***, или как RESTful, используя аннотацию ***@Path***.

Указанным Web-службам посвящены главы 5 и 6 данного учебного пособия, а сейчас проверим работоспособность EJB-компонента ***Letters***.

Для ссылок на сессионные EJB-компоненты предназначена аннотация ***EJB*** (*javax.ejb.EJB*).

В общем случае, внедрение зависимостей возможно только в управляе-

мых средах вроде EJB-контейнеров, Web-контейнеров и контейнеров клиентских приложений. Для нас важно, что внедрение EJB-компонентов можно выполнить в классе порожденном от ***HttpServlet***. Для проверки данного факта создадим в проекте ***lab4*** новый сервлет с именем ***JpaServlet*** и включим в него EJB-компоненты, показанные на листинге 3.6.

Листинг 3.6 — Тестовый вариант сервлета JpaServlet.java проекта lab4

```
package rsos.lab4;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class JpaServlet
 */
@WebServlet("/JpaServlet")
public class JpaServlet extends HttpServlet
{
	private static final long serialVersionUID = 1L;

	@EJB
	private LocalLetter	local;	// Интерфейс

	@EJB
	private RemoteLetter	remote;	// Интерфейс

	@EJB
	private Letters		letters;	// EJB-компонент

	/**
	 * Конструктор
	 */
	public JpaServlet() { super(); }

	/**
	 * Метод doGet(...)
	 */
	protected void doGet(HttpServletRequest request,
			HttpServletResponse response)
				throws ServletException, IOException
	{
		response.setCharacterEncoding("UTF-8");
		response.setContentType("text/plain");
		PrintWriter out = response.getWriter();

		out.println("local  =[" + local.getLetter(1)   + "]");
		out.println("remote =[" + remote.getLetter(2)  + "]");
		out.println("letters=[" + letters.getLetter(3) + "]");
```

```
	}
}
```

Запуск сервлета показывает одинаковую и правильную реакцию на вызов каждого EJB-компонента, как это показано на рисунке 3.2.

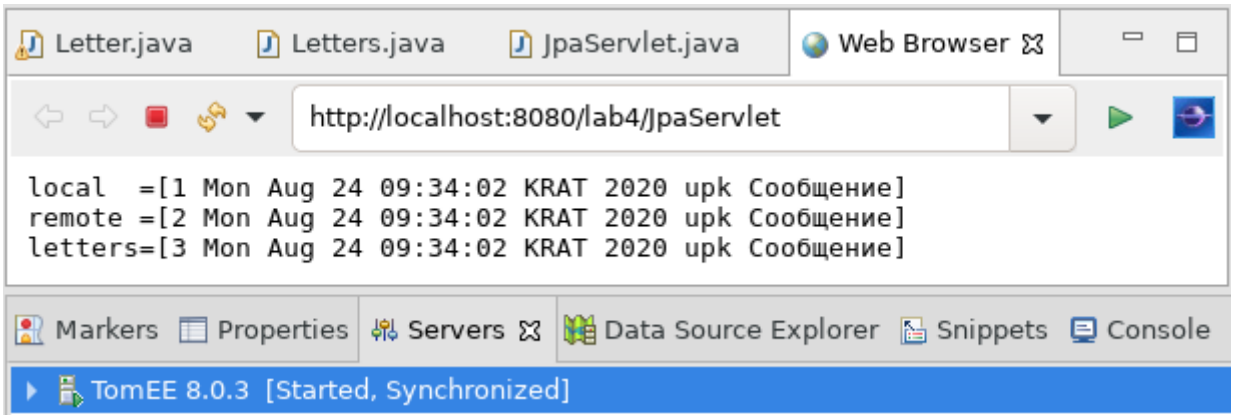

Рисунок 3.2 — Тестовый вызов трех EJB-компонентов

3.1.3 Тестовый HttpServlet проекта lab4

Для демонстрации учебного материала данной главы нам потребуется полноценная реализация сервлета ***JpaServlet***, который бы:

а) обеспечивал обращение ко всем методам EJB-компоненты ***Letters***;

б) отображал список объектов ***Letter***;

в) выводил нужную форму для взаимодействия с сервером приложений.

Для удовлетворения указанных требований, преобразуем класс ***Letters*** к первоначальному тестовому виду, показанному на листинге 3.7.

Листинг 3.7 — Тестовый вариант класса Letters.java проекта lab4

```
package rsos.lab4;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class Letters implements LocalLetter, RemoteLetter
{
	public List<Letter> getList()
	{
		List<Letter> list = new ArrayList<>();
```

```
                Letter
                letter = new Letter(new Date(), "upk", "Сообщение 1");
                letter.setId(1);
                list.add(letter);

                letter = new Letter(new Date(), "upk", "Сообщение 2");
                letter.setId(2);
                list.add(letter);

                letter = new Letter(new Date(), "upk", "Сообщение 3");
                letter.setId(3);
                list.add(letter);

                return list;
        }

        public Letter getLetter(int id) {
                return null;
        }

        public void addLetter(Letter letter) {}

        public void deleteLetter(int id) {}

        public void modLetter(Letter letter) {}
}
```

Обратите внимание, что аннотация ***@LocalBean*** обязательно должна присутствовать, иначе не будет возможен локальный доступ к классу ***Letters***.

Теперь создадим JSP-страницу ***jpa_test.jsp***, которая будет выводить результаты работы сервера и содержать форму для диалога. Также учтем, что сервлет должен передавать в JSP-страницу следующие атрибуты:

а) **"action"** — действие выбранное пользователем;

б) **"id"** — номер сообщения;

в) **"user"** — имя пользователя;

г) **"text"** — содержимое сообщения;

д) **"state"** — состояние, которое установил сервлет.

Листинг 3.8 — JSP-страница jpa_test.jsp проекта lab4

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<jsp:useBean id="letter" class="rsos.lab4.Letter" scope="page" />

<html>
```

```
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>lab4</title>
</head>
<body>
        <hr>
                  <b>Тестовая страница проекта lab4</b>
        <hr>
    <table id="table1" cellspacing="5" cellpadding="10" border="0" >
        <thead><tr>

            <th>№</th>
            <th>Дата</th>
            <th>Польз.</th>
            <th>Сообщение</th>

        </tr></thead>
        <tbody>
        <c:forEach items="${letters}" var="letter">
          <tr>
          <td><c:out value="${letter.id}"   default="*"               /></td>
          <td><c:out value="${letter.dateString}" default="Нет даты"  /></td>
          <td><c:out value="${letter.user}" default="Нет пользователя"/></td>
          <td><c:out value="${letter.text}" default="Нет текста..."   /></td>
          </tr>
        </c:forEach>
        </tbody>
    </table>
          <hr>
          <form action="JpaServlet" method="post" accept-charset="UTF-8">
          <table id="table2">
            <tr>
                  <td>Номер сообщения: </td>
                  <td><input type="text" size="6" name="id"
                        value="${id}" /></td>
            </tr>
            <tr>
                  <td>Имя пользователя: </td>
                  <td><input type="text" size="40" name="user"
                        value="${user}" /></td>
            </tr>
            <tr>
                  <td>Новый текст: </td>
                  <td><textarea rows="5" cols="40" name="text">
                  <%= request.getAttribute("text") %> </textarea></td>
            </tr>
          </table>
      <hr>
            Состояние запроса (<%= request.getAttribute("action") %>):
                              <%= request.getAttribute("state") %>
      <hr>
      <table id="table2">
      <tr>
            <td>
            <input type="radio" name="action" value="list" checked="checked">
                        Обновить
            </td>
```

```
            <td><input type="submit" value="Отправить запрос" />
            </td>
        </tr>
        <tr>
            <td>
            <input type="radio" name="action" value="add">
                        Добавить<br>
            <input type="radio" name="action" value="get">
                        Прочитать по номеру<br>
            <input type="radio" name="action" value="delete">
                        Удалить по номеру<br>
            <input type="radio" name="action" value="mod">
                        Модифицировать по номеру<br>
            </td>
        </tr>
        </table>
    </form>
    <hr>

</body>
</html>
```

Наконец, сам сервлет ***JpaServlet*** представлен на листинге 3.9, а результат его запуска — на рисунке 3.3.

Листинг 3.9 — Исходный текст сервлета JpaServlet.java проекта lab4

```
package rsos.lab4;
import java.io.IOException;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class JpaServlet
 */
@WebServlet("/JpaServlet")
public class JpaServlet extends HttpServlet
{
      private static final long serialVersionUID = 1L;

      @EJB
      private LocalLetter     local;      // Интерфейс

      @EJB
      private RemoteLetter    remote;     // Интерфейс

      @EJB
      private Letters         letters;    // EJB-компонент
      /**
       * Конструктор
       */
```

```
public JpaServlet() { super(); }

/**
 * Метод doGet(...)
 */
protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
                        throws ServletException, IOException
{
            // Переходим к методу doPost()
            doPost(request, response);
}

/**
 * Метод doPost(...)
 */
protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
                  throws ServletException, IOException
{
      /**
       * Явная установка кодировок объектов запроса и ответа.
       * Стандартная установка контекста ответа.
       */
       response.setCharacterEncoding("UTF-8");
       response.setContentType("text/html");
       request.setCharacterEncoding("UTF-8");

      // Начальные значения
      String[] state  = {"<b>Нормально...</b>", "<b>Ошибка...</b>"};
      String   action = request.getParameter("action");
      String   id     = request.getParameter("id");
      String   user   = request.getParameter("user");
      String   text   = request.getParameter("text");

      if(action == null) action   = "list";
      if(id     == null) id       = "";
      if(user   == null) user     = "asu";
      if(user.length() == 0) user = "asu";
      if(text   == null) text     = "";

      // Читаем список и передаем в JSP-страницу
      request.setAttribute("letters", letters.getList());
      request.setAttribute("state", state[0]);

      // Действия
      if("add".equals(action)) {
            request.setAttribute("state", state[1]);
      }

      if("get".equals(action)) {
            request.setAttribute("state", state[1]);
      }

      if("delete".equals(action)) {
            request.setAttribute("state", state[1]);
      }
```

```
            if("mod".equals(action)) {
                  request.setAttribute("state", state[1]);
            }
            // Установка атрибутов
            request.setAttribute("action", action);
            request.setAttribute("id",     id);
            request.setAttribute("user",   user);
            request.setAttribute("text",   text.trim());

            /**
            * Стандартное подключение ресурса JSP-страницы
            */
            request.getRequestDispatcher("/WEB-INF/jpa_test.jsp")
                  .forward(request, response);
      }
}
```

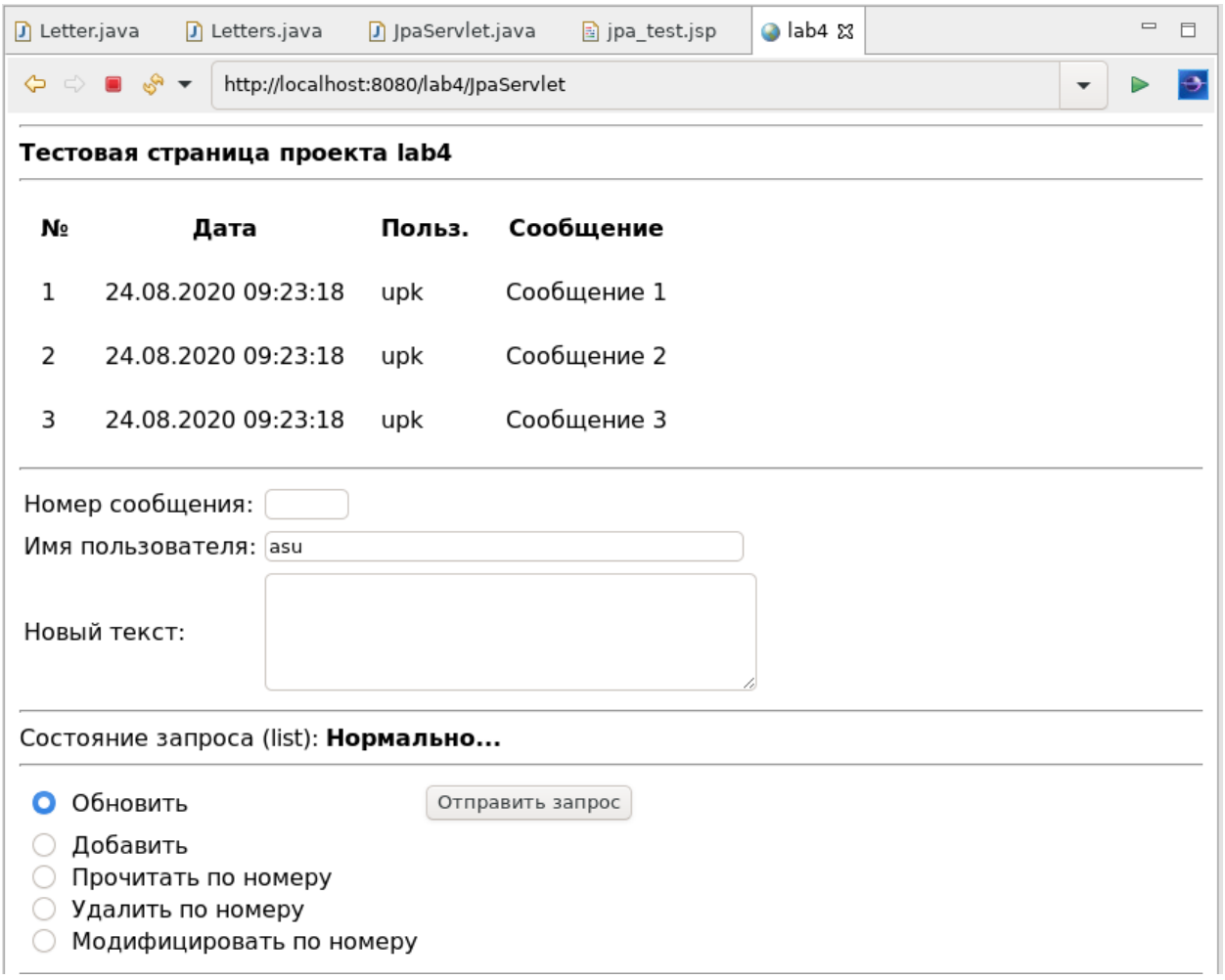

Рисунок 3.3 — Результат запуска сервлета JpaServlet

Теперь мы готовы для демонстрации учебных примеров данной главы.

3.1.4 Инфраструктура сервера TomEE и СУБД Derby

Все учебные примеры данной главы будут запускаться в проекте ***lab4*** среды разработки Eclipse EE и взаимодействвоать с СУБД Apache Derby.

Приложениям платформы Java для работы с СУБД требуется соответствующий драйвер JDBC. Для того, чтобы сервер мог устанавливать сетевое соединение с СУБД Derby, необходим файл ***derbyclient.jar***, который следует поместить в каталог ***WEB-INF/lib***.

Для того чтобы приложение проекта ***lab4*** могло соединиться с учебной базой данных, СУБД Apache Derby должен быть запущен и заданы все его параметры и дескрипторы, указанные ниже:

1) **localhost:1257** — адрес и порт, по которму должна быть запущена и доступна для сервера приложений СУБД Apache Derby;
2) **lab4db** — имя базы данных, с которой работает приложение;
3) **RSOSDB** — имя владельца и схемы базы данных;
4) **pswdb** — пароль владельца схемы базы данных;
5) **resources.xml** — дескриптор развертывания ресурсов сервера приложений Apache TomEE, который следует поместить в каталог ***WEB-INF*** проекта ***lab4***;
6) **persistence.xml** — дескриптор развертывания средств технологии JPA, который следует поместить в каталог ***WEB-INF*** проекта ***lab4***.

Учитывая, что инфраструктуры распределенных систем могут быть разными и по разному настраиваемыми, ограничимся приведенными выше данными как конкретным примером, используемым при изложении дальнейшего учебного материала.

Дескриптор развертывания ресурсов сервера приложений — это XML-файл (файлы), в которых описываются настройки контейнеров сервера, менеджеров различных технологий и ресурсов, включая ресурсы используемых СУБД — ресурс ***DataSource***.

Для сервера Apache TomEE используются следующие файлы дескрипторов развертывания ресурсов:

а) ***conf/tomee.xml*** — основной файл описания ресурсов, указанный относительно каталога развертывания дистрибутива сервера, содержимое которого распространяется на все приложения (проекты) сервера;

б) ***WEB-INF/resources.xml*** — файл описания ресурсов, расположенный в инфраструктуре конкретного проекта (в нашем случае проекта ***lab4***), содержимое которого распространяется на конкретный проект сервера.

Какой использовать файл, решает администратор сервера, а мы будем

рассматривать файл ***resources.xml***, поскольку он влияет только на один конкретный проект и настроен на нашу учебную тему, что показано на листинге 3.10.

Листинг 3.10 — Исходный текст файла resources.xml проекта lab4

```
<?xml version="1.0" encoding="UTF-8"?>

<resources>

	<Resource id="lab4Derby" type="DataSource">
		JdbcDriver = org.apache.derby.jdbc.ClientDriver
		JdbcUrl = jdbc:derby://localhost:1527/lab4db
		Username = RSOSDB
		Password = pswdb
		JtaManaged = true
	</Resource>

	<Resource id="lab4DerbyUnmanaged" type="DataSource">
		JdbcDriver = org.apache.derby.jdbc.ClientDriver
		JdbcUrl = jdbc:derby://localhost:1527/lab4db
		Username = RSOSDB
		Password = pswdb
		JtaManaged = false
	</Resource>

</resources>
```

Следует обратить внимание, что на листинге представлены два ресурса, имеющих одинаковый тип ***DataSource***, указывающий на отношение ресурса к СУБД, но разные идентификаторы ***lab4Derby*** и ***lab4DerbyUnmanaged***, указывающие на разное их содержимое.

Фактически оба ресурса относятся к СУБД Apache Derby и указывают одинаковые: драйвер СУБД, URI-адрес соединения с базой данных, а также имя и пароль владельца схемы используемой базы данных. Отличаются они только использованием менеджера транзакций, определяемым логическим параметром ***JtaManaged***.

Указанное выше описание ресурсов СУБД используется дескриптором развертывания средств технологии JPA ***persistence.xml***, содержимое которого представлено на листинге 3.11.

Листинг 3.11 — Исходный текст файла persistence.xml проекта lab4

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

     <persistence-unit name="lab4-unit1" transaction-type="RESOURCE_LOCAL">

          <non-jta-data-source>lab4DerbyUnmanaged</non-jta-data-source>
```

```
        <class>rsos.lab4.Letter</class>

    </persistence-unit>

    <persistence-unit name="lab4-unit2" transaction-type="JTA">

        <jta-data-source>lab4Derby</jta-data-source>
        <non-jta-data-source>lab4DerbyUnmanaged</non-jta-data-source>

        <class>rsos.lab4.Letter</class>

    </persistence-unit>
</persistence>
```

В этом файле приводится две единицы описания («***юниты***») средств технологии JPA, именуемые ***lab4-unit1*** и ***lab4-unit2***. Эти имена используются EJB-компонентами для указания:

а) типов транзакций: ***RESOURCE_LOCAL*** или ***JTA***;

б) имен используемых ресурсов: ***lab4Derby*** или ***lab4DerbyUnmanaged***;

в) полное имя ***rsos.lab4.Letter*** **—** класса сущности, используемой для хранения инфомационного объекта (см. листинг 3.1).

> В общем случае, содержимое файлов ***resources.xml*** и ***persistence.xml*** может существенно меняться в зависимости от используемого сервера приложений, типа СУБД и провайдеров технологии JPA.

Учитывая указанную ситуацию, листинги 3.11 и 3.12 конкретно адаптированы для учебного процесса изучаемой дисциплины. В условиях других программно-аппаратных средств распределенных систем, студенту придется использовать конкретную специальную литературу.

> Из какой директории запущен сетевой вариант Apache Derby, в том каталоге он будет создавать базы данных и управлять ими.

Поскольку используемая в данном учебном процессе технология JPA не обеспечивает успешное самостоятельное создание баз данных, то они должны быть заранее созданы специальными сценариями на языке SQL.

В пределах учебного процесса, для работы с СУБД Apache Derby студенту выделен каталог ***rsosDB*** рабочей области пользователя ***upk*** (см. учебно-методическое пособие [21]). Содержимое этого каталога показано на рисунке 3.4:

а) ***lab4db*** — каталог размещения учебной базы данных;

б) ***create_lab4db.sql*** — SQL-сценарий создания базы данных ***lab4db***;

в) ***select_lab4db.sql*** — SQL-сценарий чтения содержимого базы ***lab4db***;

г) ***startServerDB*** — командный сценарий запуска сетевого варианта СУБД;

д) ***stopServer*** — командный сценарий оснановки СУБД.

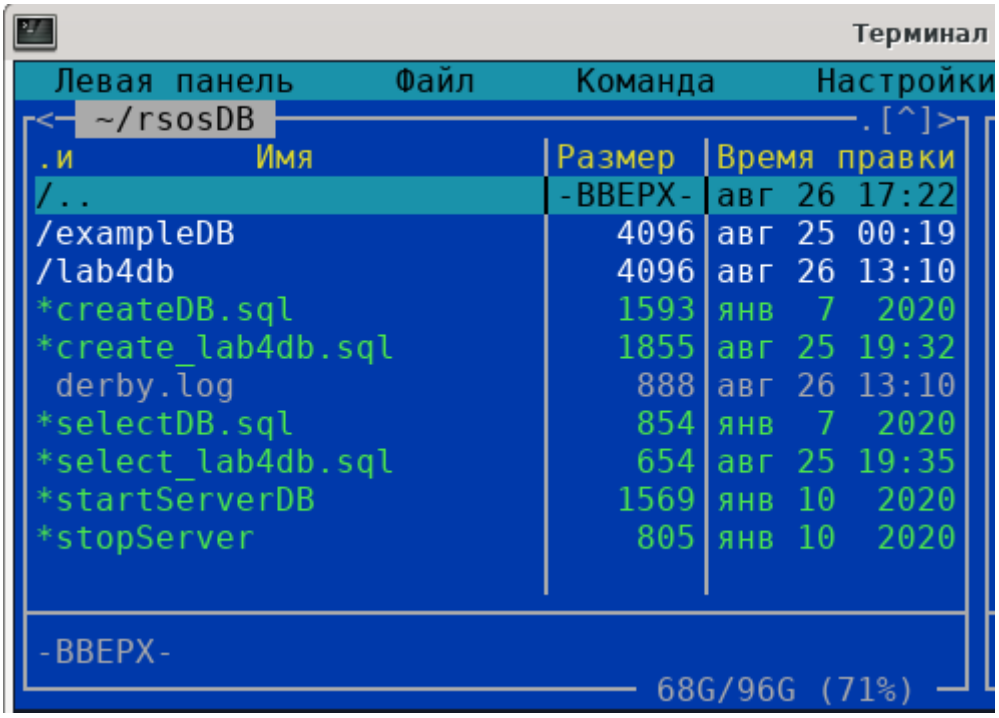

Рисунок 3.4 — Содержимое каталога rsosDB для работы с СУБД Derby

Поскольку правильная работа технологии JPA зависит о правильной структуры базы данных и используемой таблицы, на рисунке 3.5 представлено содержимое SQL-сценария ***create_lab4db.sql***.

Сначала запускается Apache Derby из каталога ***rsosDB***, а только потом стартуется сервлет ***JpaServlet*** проекта ***lab4***.

Сервлет ***JpaServlet*** будет инкасулировать EJB-компоненты, которые будут автоматически подключаться к СУБД при активации. Поэтому, если СУБД не запущена, то сервер Apache TomEE может не запуститься, а если по каким-либо причинам база данных ***lab4db*** «*сломалась*», то следует:

а) ***остановить*** СУБД Apache Derby;

б) ***удалить*** каталог lab4db;

в) ***создать*** новую базу данных командой: **`ij create_lab4db.sql`**;

г) ***запустить*** СУБД Apache Derby.

Файл Правка Поиск Вид Документ Справка

```
1  ------------------------------------------------------------
2  -- Сценарий создания базы данных: lab4db.
3  -- Создается схема для JpaServlet.
4  ------------------------------------------------------------
5  -- Вариант сервера:
6   CONNECT 'jdbc:derby://localhost:1527/lab4db;create=true;'
7          USER 'RSOSDB' PASSWORD 'pswdb';
8  ------------------------------------------------------------
9  -- Создаем схему
10 CREATE SCHEMA RSOSDB AUTHORIZATION pswdb;
11 -- Создаем последовательность
12 CREATE SEQUENCE SEQUENCE
13 AS INTEGER
14 START WITH 1;
15 ------------------------------------------------------------
16 -- Сначала удаляем все таблицы, если они были созданы.
17 drop table t_letter;
18 ------------------------------------------------------------
19 -- Создаем таблицу notepad
20
21 create table t_letter (
22         -- ключ записи - ключ таблицы
23         ID integer NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
24         DATE timestamp NOT NULL,
25         NAME varchar(255),
26         TEXT varchar(4096),     -- текст записи
27         primary key (ID)
28 );
29 ------------------------------------------------------------
30 -- Вносим в таблицу некоторое количество записей:
31 insert into t_letter(DATE, NAME, TEXT) values(CURRENT_TIMESTAMP, 'vgr', 'Запись 1');
32 insert into t_letter(DATE, NAME, TEXT) values(CURRENT_TIMESTAMP, 'asu', 'Запись 2');
33 insert into t_letter(DATE, NAME, TEXT) values(CURRENT_TIMESTAMP, 'upk', 'Запись 2');
34 ------------------------------------------------------------
35 select * from t_letter;
36
37 ------------------------------------------------------------
38 -- Завершение работы сценария:
39 commit;
40 disconnect;
41 exit;
42 ------------------------------------------------------------
```

Рисунок 3.5 — SQL-сценарий для создания базы данных lab4db

SQL-сценарий ***create_lab4db.sql*** вставляет в таблицу ***t_letter*** три новых записи, которые можно использовать для первоначального тестирования EJB-компонент. Следует также изучить структуру и поля используемой таблицы, поскольку она используется в теоретическом материале следующего подраздела.

3.2 Технология JPA

Технология JPA (*Java Persistence API*) является наиболее современной и развиваемой технологией программной платформы Java EE.

Проблема современного программирования приложений, работающих с базами данных, состоит в постоянном преобразовании данных, содержащихся в объектах, в представления табличных данных, хранящихся под управлением той или иной СУБД.

Идея технологии JPA состоит во взаимодействии с СУБД на уровне объектов (*сущностей, entity*), предоставляя программистам максимальное использование языка Java. Это достигается тремя теоретическими положениями, которые мы кратко рассмотрим в данном подразделе:

а) ***формализацией*** понятия сущности и обеспечение ее описания адекватными средствами Java EE;

б) ***отображением*** класса сущности на релляционное представление ORM (*Object-Relational Mapping*);

в) ***концепцией*** менеджера сущностей и языка запросов JPQL (*Java Persistence Query Language*).

Посколку тема технологии JPA — достаточно емкая и не является основной темой нашей дисциплины, то изложенный учебный материал демонстрирует только основные ее моменты, достаточные для понимания самой идеи и реализации тестового примера, заявленного в предыдущем подразделе.

3.2.1 Сущности

Сущности (***Entity***) — это специальным образом аннотированные JAVA-классы, воспринимаемые контейнерами платформы Java EE.

Формально, ***класс-сущность*** должен удовлетворять следующим ограничениям:

1. Класс должен быть снабжен аннотацией ***@javax.persistence.Entity*** или быть обозначен в XML-дескрипторе как сущность.
2. Для обозначения простого первичного ключа должна быть использована аннотация ***@javax.persistence.Id***.
3. Класс должен располагать конструктором без аргументов, который должен быть ***public*** или ***protected***, но также может иметь другие конструкторы с аргументами.
4. Должен быть классом верхнего уровня. Перечисление или интерфейс не

могут быть обозначены как сущность.

5. Не должен быть ***final*** и ни один из методов или постоянные переменные экземпляра класса-сущности тоже не могут быть ***final***.
6. Если экземпляр сущности будет передаваться с использованием значения как обособленный объект, например, с помощью удаленного интерфейса, то он должен реализовывать интерфейс ***Serializable***.

Фактически, класс-сущность — это ***сериализованный POJO-класс***, который расширен бизнес-методами до потребностей приложений.

Чтобы не быть голословным, рассмотрим класс ***Letter***, представленный ранее на листинге 3.1, и преобразуем его в сущность двумя аннотациями, как это показано на листинге 3.13.

Листинг 3.13 — Преобразование класса Letter.java проекта lab4 в сущность

```
package rsos.lab4;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    @Id
    private int id;

    private Date date;
    private String name;
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
```

Первая аннотация ***@Entity*** указывает, что класс ***Letter*** является сущностью и будет отображаться в таблицу базы данных с именем ***LETTER***.

Вторая аннотация ***@Id*** указывает, что целочисленный атрибут ***id*** является уникальным ключем таблицы ***LETTER***.

Дальнейшие модификации сущности ***Letter*** определяются многочисленными аннотациями объектно-реляционного отображения.

3.2.2 Объектно-реляционное отображение

Теоретически, технология JPA позволяет средствами ORM создать все необходимые таблицы базы данных по описанию аннотированной сущности.

На практике, возможности ORM ограничены возможностями провайдера JPA и особенностями конкретной СУБД.

Для СУБД Derby отображение простейших объектных типов представлено в таблице 3.1.

Таблица 3.1 — Отображение типов языка Java в типы СУБД Derby

Тип языка Java	***Тип СУБД Derby***
Boolean	SMALLINT
Integer	INTEGER
Long	BIGINT
Float	FLOAT
String	VARCHAR
Double	DOUBLE

Часто разработчику приложений требуется модифицировать уже существующие сущности к уже существующим таблицам базы данных. Например, SQL-сценарий ***create_lab4db.sql***, представленный на рисунке 3.5, создает учебную таблицу ***t_letter***. Чтобы полностью согласовать сущность ***Letter*** и существующую базу данных ***lab4db***, следует использовать аннотации:

а) ***@Table(name = "t_letter")*** — изменяет имя используемой таблицы;

б) ***@GeneratedValue(strategy = GenerationType.IDENTITY)*** — задает стратегию генерации первичного ключа.

С учетом сделанных замечаний, полностью адаптированная для работы с учебной базой данных сущность ***Letter*** будет иметь заголовочную часть, показанную на листинге 3.14.

Листинг 3.14 — Полностью адаптированная сущность Letter.java

```
package rsos.lab4;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
import javax.persistence.Table;

@Entity
@Table(name = "t_letter")
public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private Date date;
    private String name;
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
```

В общем случае, ***первичные ключи*** сущностией могут быть составными. Тогда они оформляются в виде отдельных классов, а затем обозначаются в теле сущностей с помошью пары аннотаций ***@Embeddable/@EmbeddedId*** или одной аннотацией ***@IdClass***, в зависимости от используемого метода.

Мы не будем подробно рассматривать тему первичных ключей, а интересующихся отправляем, например, к литературному источнику [17]. Мы также не будем рассматривать многие другие полезные аннотации, а оганичимся только тремя: @Column, @Temporal и @Transient.

@javax.persistence.Column — аннотация, позволяющая изменять свойства атрибутов сущности при отражении в столбцы таблицы базы данных. Ее формальное определение представлено на листинге 3.15.

Листинг 3.15 — Полностью адаптированная сущность Letter.java

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
      String name() default "";          // имя столбца
      boolean unique() default false;    // уникальность
      boolean nullable() default true;
      boolean insertable() default true;
      boolean updatable() default true;
      String columnDefinition() default "";
      String table() default "";
      int length() default 255;          // длина по умолчанию
      int precision() default 0;         // десятичная точность
      int scale() default 0;             // десятичная система счисления
}
```

Она позволяет изменять отражаемое имя столбца таблицы, делать его

уникальным, ненулевым, неадаптируемым и многое другое. Например, если мне необходимо, чтобы в таблице ***t_letter***: столбец ***DATE*** не принимал значение ***null***; атрибут сущности ***name*** отображался в столбец ***USER***; столбец ***TEXT*** имел размер ***4096*** символов (не более 32.672 символа), тогда содержимое листинга 3.14 следует заменить на содержимое листинга 3.16.

Листинг 3.16 — Сущность Letter.java с аннотациями @Column

```
@Entity
@Table(name = "t_letter")
public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
          private Date date;
    @Column(name = "USER")
    private String name;
    @Column(length = 4096)
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
```

@javax.persistence.Temporal — аннотация, позволяющая правильно отображать JAVA-типы ***java.util.Date*** и ***java.util.Calendar*** в типы ***java.sql.Date*** (только дата), ***java.sql.Time*** (только время) и ***java.sql.Timestamp*** (дата и время), которыми пользуется СУБД Apache Derby. Эта аннотация использует константы: ***TemporalType.DATE***, ***TemporalType.TIME*** и ***TemporalType.TIMESTAMP***. Например, если учесть, что столбец ***DATE*** таблицы ***t_letter*** определен как ***timestamp*** (см. рисунок 3.5), то листинг 3.16 следует переписать в виде, представленном листингом 3.17.

Листинг 3.17 — Сущность Letter.java с аннотацией @Temporal

```
@Entity
@Table(name = "t_letter")
public class Letter implements Serializable
{
      // Идентификатор сериализации
      private static final long serialVersionUID = 1L;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
```

```
    @Column(nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    private String name;
    @Column(length = 4096)
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
```

@javax.persistence.Transient — аннотация, позволяющая отмечать те атрибуты сущности, которые не должны отражаться в таблице базы данных.

Например, если предположить, что в таблице ***t_letter*** отсутствует столбец с именем ***NAME***, то сущность ***Letter.java*** должна быть описана, как показано на листинге 3.18.

Листинг 3.18 — Сущность Letter.java с аннотацией @Transient

```
@Entity
@Table(name = "t_letter")
public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Date date;
    @Transient
    private String name;
    @Column(length = 4096)
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
```

Подводя итог краткому описанию объектно-реляционного отображения, отметим, что имеется множество аннотаций, позволяющих достаточно точно описать содержимое отдельных таблиц базы данных и их связи.

> Наиболее правильным является отображение таблицы ***t_letter***, которое представленно на листинге 3.17.

3.2.3 Манеджер сущностей

Центральным элементом JPA-технологии является менеджер сущностей, определяемый классом ***javax.persistence.EntityManager***, описывающий API-интерфейс взаимодействия с ними и отвечающий за управление этими сущностями.

В реальности, ***EntityManager*** — это всего лишь интерфейс, реализация которого обеспечивается поставщиком (*провайдером*) JPA, например, таким как EclipseLink. Далее мы не будем различать эти тонкости, говоря о менеджере как о действительном исполнителе.

EntityManager — это средство взаимодействия с базой данных для выполнения:

а) ***простейших операций*** CRUD (*Create, Read, Update и Delete*);

б) ***сложных запросов*** с помощью языка JPQL (*Java Persistence Quary Language*).

Чтобы получить для управления объект менеджера сущностей, необходимо сначала обратиться к фабрике ***EntityManagerFactory***, как это показано на листинге 3.19.

Листинг 3.19 — Команды создания объекта типа EntityManager

```
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("lab4-unit1");
EntityManager em =
      emf.createEntityManager();
```

Обратите внимание, что метод фабрики в качестве аргумента использует ссылку на юнит «***lab4-unit1***». В результате, идет обращение к дескриптору, заданному файлом ***persistence.xml*** (см. листинг 3.11). Далее, по имени ***lab4DerbyUnmanaged***, идет обращение к файлу ресурса ***resources.xml*** (см. листинг 3.10), где содержится информация о параметрах соединения с базой данных. Таким образом, объект менеджера сущностей ***em*** может быть подключен к нужной базе данных.

Если внимательно рассмотреть файл ***persistence.xml***, то мы увидим описание второго юнита «***lab4-unit2***», как это показано ниже.

```
 <persistence-unit name="lab4-unit1" transaction-type="RESOURCE_LOCAL">
      <!-- Описание юнита не-JTA-->
</persistence-unit>

 <persistence-unit name="lab4-unit2" transaction-type="JTA">
      <!-- Описание юнита JTA-->
</persistence-unit>
```

Принципиальная разница между ними состоит в том, что они поддерживают разный тип тпранзакций:

а) тип транзакции «***RESOURCE_LOCAL***» — без автоматического обеспечения транзакций контейнером; требует, чтобы программист самостоятельно указывал начало и конец транзакций;

б) тип транзакции «***JTA***» (*Java Tramsaction API*) — автоматическое обеспечение транзакций контейнером; не требует вмешательства программиста.

Излишне утверждать, что объект типа ***EntityManager*** обладает большим количеством различных методов.

В таблице 3.2 представлен набор некоторых важных методов, вполне достаточных для первоначального изучения и демонстрации учебных примеров.

Таблица 3.2 — Перечень некоторых важных методов менеджера сущностей

Метод	***Описание метода***
void persist(Object entity)	Запись сущности в базу данных.
<T> T find(Class<T> entityClass, Object primaryKey)	Выполняет поиск объекта сущности по первичному ключу таблицы.
<T> T getReference(Class<T> entityClass, Object primaryKey)	Получает ссылку на объект сущности по первичному ключу таблицы.
void remove(Object entity)	Удаляет объект сущности из базы данных.
void refresh(Object entity)	Восстанавливает измененый объект сущности по значению сохраненному в базе данных.
void flush()	Принудительно сбрасывает объект сущности из кэша JPA в базу данных.
void clear()	Отключает все сущности от от управления менеджером.
void detach(Object entity)	Отключает отдельную сущность от от управления менеджером.
boolean contains(Object entity)	Проверяет, отключена ли отдельная сущность от от управления менеджером.
EntityTransaction getTransaction()	Возвращает объект транзакций управляемых сущностей. Метод ***begin()*** - начинает транзакцию, а метод ***commit()*** - завершает ее, ***rollback()*** - отмена транзакции.
<T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass)	Создает типизированный объект запроса к базе данных. Далее, метод ***getResultList()*** - возвращает список объектов, а метод ***getSingleResult()*** - возвращает одиночный экземпляр объекта или исключение ***NonUniqueResultException***.

Прежде чем переходить к примеру, обратите внимание, что все методы менеджера сущностей используют в качестве своих аргументов объекты, а внашей базе данных уникальный ключ и используемой сущности ключ ***id*** является простым типом ***int***. Поэтому, чтобы не усложнять листинги преобразованиями типов, создадим и будем использовать простой класс ***ToInt***, представленный на листинге 3.20.

Листинг 3.20 — Исходный текст класса ToInt.java проекта lab4

```
	package rsos.lab4;

public class ToInt {
	/**
	 * Возвращает объект Integer или null
	 */
	public static Integer get(String ss) {
		if(ss == null)
				return null;
		try {
			return Integer.decode(ss);

		} catch(NumberFormatException e) {

				return null;
		}
	}
}
```

Обратите внимание, что единственный метод ***get(...)*** этого класса является статическим, преобразует строку символов в объект типа ***Integer*** или значение ***null***, в противном случае.

3.2.4 Пример использования не-JTA-типа транзакций

Stateless EJB-контейнер способен инкапсулировать объект фабрики менеджера сущностей и использовать ***EntityManager*** для не-JTA-типа транзакций.

Применение менеджера сущностей продемострируем на примере EJB-класса ***Lets1.java***, который обеспечивает чтение списка записей из учебной базы данных.

За основу этого класса возьмем содержимое класса ***Letters.java*** (см. листинги 3.4 и 3.5), в котором изменим имя и реализуем всего один метод ***getList()***, как это показано на листинге 3.21.

Такой подход позволит сохранить старый листинг EJB-класса и уменьшит путаницу с различными реализованными версиями примеров.

Листинг 3.21 — Исходный текст EJB-компонента Lets1.java

```
package rsos.lab4;
import java.util.List;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.PersistenceUnit;
import javax.persistence.TypedQuery;

@Stateless
@LocalBean
public class Lets1 implements LocalLetter, RemoteLetter
{
    // Подключаем объект менеджера фабрики
    @PersistenceUnit(name = "lab4-unit1")
    private EntityManagerFactory emf;

    // Метод чтения содержимого таблицы t_letter
    public List<Letter> getList()
    {
      // Создаем объект менеджера сущностей
      EntityManager em =
            emf.createEntityManager();

      // Создаем объект менеджера транзакций
      EntityTransaction tr =
            em.getTransaction();

      // Начинаем транзакцию
      tr.begin();

      // Создаем типизированный объект запроса
      TypedQuery<Letter> query =
            em.createQuery("SELECT c FROM Letter c", Letter.class);

      // Делаем запрос и получаем список объектов
      List<Letter> list =
            query.getResultList();

      // Завершаем транзакцию
      tr.commit();
      return list;
    }

    public Letter getLetter(int id) {
         return null;
    }

    public void addLetter(Letter letter) {}

    public void deleteLetter(int id) {}

    public void modLetter(Letter letter) {}
}
```

В приведенном листинге, подключение объекта мененджера фабрики выделено серым цветом. Далее, в каждом реализуемом методе следует выполнить ряд стандартных шагов:

1) создать объекты менеджера сущностей ***em*** и менеджера транзакций ***tr***;
2) выполняемые действия с базой данных заключить в команды транзакций: ***tr.begin()*** и ***tr.commit()***;
3) выполнить сами действия с базой данных; в нашем случае — это создание объекта запроса ***query*** и получение рзультата запроса в виде списка ***list***.

Обратите внимание, что в самом методе формирования запроса (метод ***createQuery(…)***) оператор ***SELECT*** обращается не к таблице базы данных, а к классу ***Letter***.
Заметьте, что для EJB-компоненты ***Lets1*** будет создан отдельный контейнер типа «***STATELESS***», который и обеспечивает работу всех менеджеров.

Для демонстрации работы созданной EJB, откроем новый сервлет с именем ***JpaServlet1.java***. Изменения, которые внесены в исходный текст предыдущего сервлета, выделены серым цветом и показаны на листинге 3.22.

Листинг 3.22 — Исходный текст сервлета JpaServlet1 проекта lab4

```
package rsos.lab4;

import java.io.IOException;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class JpaServlet1
 */
@WebServlet("/JpaServlet1")
public class JpaServlet1 extends HttpServlet
{
    private static final long serialVersionUID = 2L;

    // Инкапсуляция EJB-компонента типа Lets1
    @EJB
    private Lets1             lets1;      // EJB-компонент

    /**
     * Конструктор
     */
    public JpaServlet1() { super(); }

    /**
```

```
 * Метод doGet(...)
 */
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
                        throws ServletException, IOException
{
  // Переходим к методу doPost()
  doPost(request, response);
}

/**
 * Метод doPost(...)
 */
protected void doPost(HttpServletRequest request,
              HttpServletResponse response)
                    throws ServletException, IOException
{
  /**
   * Явная установка кодировок объектов запроса и ответа.
   * Стандартная установка контекста ответа.
   */
  response.setCharacterEncoding("UTF-8");
  response.setContentType("text/html");
  request.setCharacterEncoding("UTF-8");

  // Начальные значения
  String   state  = "<b>Нормально...</b>";
  String   action = request.getParameter("action");
  String   sId    = request.getParameter("id");
  Integer  Id     = ToInt.get(sId);
  String   user   = request.getParameter("user");
  String   text   = request.getParameter("text");

  if(action == null) action   = "list";
  if(user   == null) user     = "asu";
  if(user.trim().length() == 0) user = "asu";
  if(text   == null) text     = "";

  request.setAttribute("state", state);

  // Действия
  if("add".equals(action)) {
              text = "";
  }

  if("get".equals(action)) {
        if(Id == null) {
              request.setAttribute("state",
                    "<b>Ошибочный идектификатор сообщения...</b>");
              text = "";
        }
  }

  if("delete".equals(action)) {
        if(Id == null) {
              request.setAttribute("state",
                    "<b>Ошибочный идектификатор сообщения...</b>");
```

```
                text = "";
            }
        }

        if("mod".equals(action)) {
            if(Id == null) {
                request.setAttribute("state",
                    "<b>Ошибочный идектификатор сообщения...</b>");
            }
        }

        request.setAttribute("lists1",  lets1.getList());

        // Установка атрибутов
        request.setAttribute("action",  action);
        request.setAttribute("id",      sId);
        request.setAttribute("user",    user);
        request.setAttribute("text",    text.trim());

        /**
         * Стандартное подключение ресурса JSP-страницы
         */
        request.getRequestDispatcher("/WEB-INF/jpa_test1.jsp")
                .forward(request, response);
    }
}
```

Наиболее важными изменениями, внесенными в исходный текст сервлета ***JpaServlet1***, являются:

а) инкапсуляция EJB-компонента ***Lets1***;

б) подключение новой JSP-страницы ***jpa_test1.jsp***;

в) передача в JSP-страницу ***jpa_test1.jsp*** нового атрибута ***lists1***.

Новая JSP-страница показана на листинге 3.23, а результат запуска сервлета показан на рисунке 3.6.

Листинг 3.23 — Исходный текст JSP-страницы jpa_test1.jsp проекта lab4

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                "http://www.w3.org/TR/html4/loose.dtd">

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

<jsp:useBean id="letter1" class="rsos.lab4.Letter" scope="page" />

<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>lab4</title>
</head>
<body>
    <hr>
```

```
      <b>Тестовая страница проекта lab4 (jpa_test1.jsp)</b>
      <hr>
      <table id="table1" cellspacing="0" cellpadding="5" border="0" >
      <thead><tr>

            <th align="left" >№</th>
            <th align="left" >Дата</th>
            <th align="left" >Пользователь</th>
            <th align="left" >Сообщение</th>

      </tr></thead>
      <tbody>
      <c:forEach items="${lists1}" var="letter1">
      <tr>
      <td><c:out value="${letter1.id}" default="*"           /></td>
      <td><c:out value="${letter1.dateString}" default="Нет даты"  /></td>
      <td><c:out value="${letter1.name}" default="Нет пользователя"/></td>
      <td><c:out value="${letter1.text}" default="Нет текста..."   /></td>
      </tr>
      </c:forEach>

      </tbody>
      </table>
      <hr>
      <form action="JpaServlet1" method="post" accept-charset="UTF-8">
      <table id="table2">
            <tr>
                  <td>Номер сообщения: </td>
                  <td><input type="text" size="6" name="id"
                        value="${id}" /></td>
            </tr>
            <tr>
                  <td>Имя пользователя: </td>
                  <td><input type="text" size="40" name="user"
                        value="${user}" /></td>
            </tr>
            <tr>
                  <td>Новый текст: </td>
                  <td><textarea rows="5" cols="40" name="text">
                        <%= request.getAttribute("text") %> </textarea></td>
            </tr>
      </table>
      <hr>
Состояние запроса (<b><i><%= request.getAttribute("action")%></i></b>):
                   <b><i><%= request.getAttribute("state") %></i></b>
    <hr>
    <table id="table3">
       <tr>
            <td>
            <input type="radio" name="action" value="list" checked="checked">
                        Обновить
            </td>
            <td><input type="submit" value="Отправить запрос" />
            </td>
       </tr>
       <tr>
          <td>
```

```
            <input type="radio" name="action" value="add">
                    Добавить<br>
            <input type="radio" name="action" value="get">
                    Прочитать по номеру<br>
            <input type="radio" name="action" value="delete">
                    Удалить по номеру<br>
            <input type="radio" name="action" value="mod">
                    Модифицировать по номеру<br>
      </td>
      </tr>
    </table>
    </form>
    <hr>

</body>
</html>
```

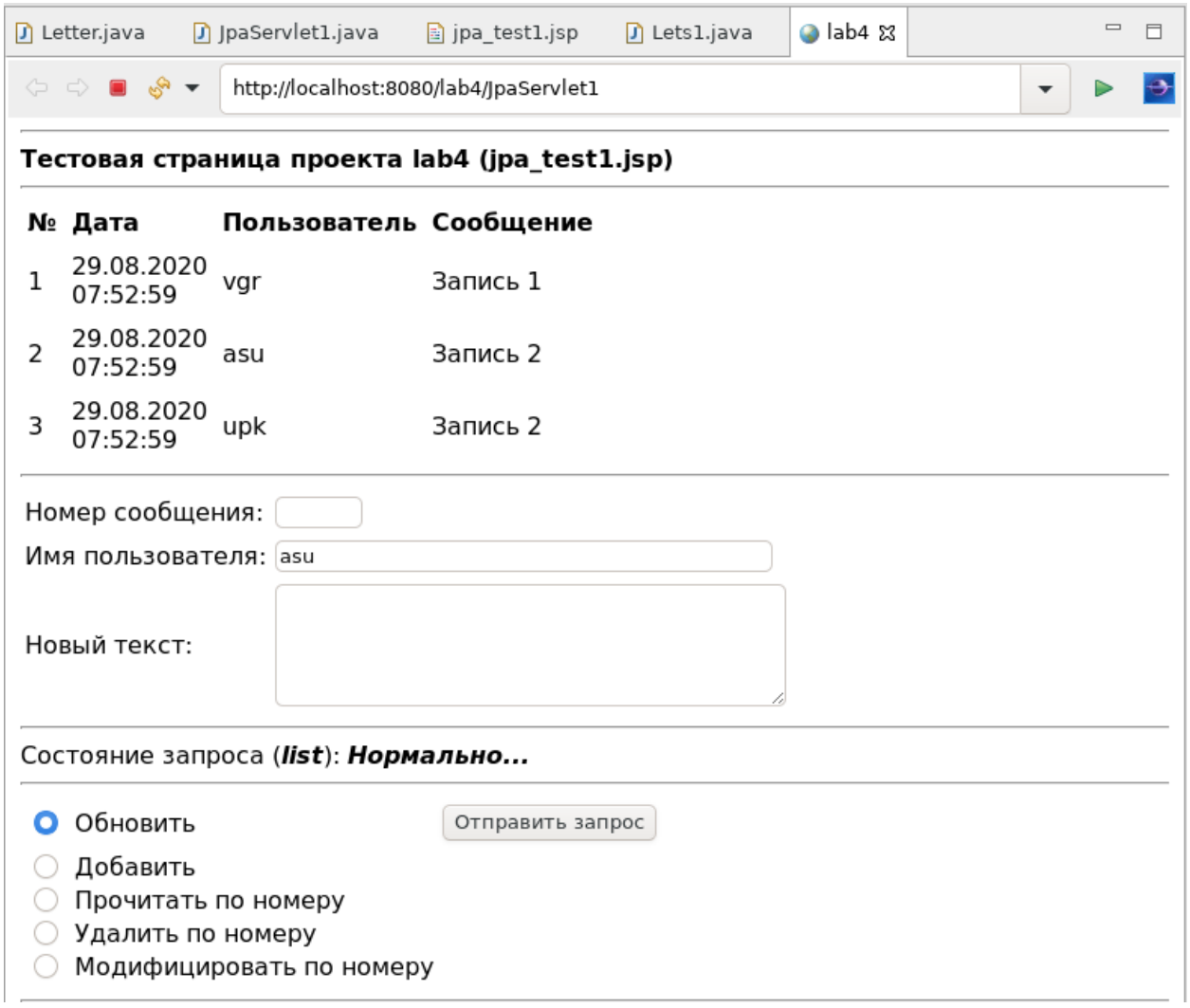

Рисунок 3.6 — Результат запуска сервлета JpaServlet1

3.3 Транзакции управляемые контейнером

Наиболее важные направления современного развития технологии JPA связаны с транзакциями управляемыми контейнерами JTA и объектно-ориентированным языком запросов JPQL.

В предыдущем подразделе технология JPA была продемонстрирована на простом примере чтения всех записей объектов сущности ***Letter*** из таблицы ***t_letter*** базы ***lab4db***. При этом использовался тип транзакций не-JTA и простая форма запроса языка JPQL.

Несмотря на максимальную простоту использованного примера, использованные методы имеют ряд принципиальных недостатков:

- **тип транзакций не-JTA**, заданный в определении юнита «***lab4-unit1***» как параметр ***transaction-type="RESOURCE_LOCAL"***, требует использование фабрики менеджера сущностей и «*ручное*» управление транзакциями;
- **простая форма JPQL-запроса** требует явного использование языка SQL и ошибки, которые допустил программист, будут обнаружены только во время выполнения самого запроса.

Учебная цель данного подраздела — демонстрация использования объектных JPQL-запросов ***Criteria API*** и менеджера сущностей, использующего тип транзакций контейнера ***transaction-type="JTA"***.

Демонстрационный характер изложения учебного материала обоснован большим объемом технологических решений рассматриваемого направления и стремлением завершить полную реализацию JPA-сервлета.

3.3.1 Объектно-ориентированные запросы Criteria API

Язык JPQL (*Java Persistence Query Language*) обеспечивает пять типов запросов.

Формулировка целевых действий, осуществляемых программистом с использованием технологии JPA, обеспечивается пятью типами JPQL-запросов:

1. ***Динамические запросы*** — простейшая форма JPQL-запроса, формулируемая в виде текстовой строки и динамически генерируемая во время его выполнения. Такой тип запроса был использован в тестовом примере предыдущего подраздела (см. пункт 3.2.4, листинг 3.21).
2. **Именованные запросы** — форма JPQL-запросов, формулируемая в виде текстовых строк во время описания класса сущности с помощью аннота-

ций ***@NamedQueries*** и ***@NamedQuery***. В отличие от динамических запросов, они являются ***статическими*** (*неизменяемыми*), поскольку генерируются во время компиляции класса сущности.

3. **Criteria API** — полностью объектно-ориентированная концепция запросов Query API, исключающая применение строковых операторов SQL-подобных языков.
4. **Родные запросы** — запросы, которые вместо JPQL-операторов, формулируются на основе «*родных*» SQL-операторов SELECT, UPDATE или DELETE. В частности, они могут быть именованными запросами, для чего используется аннотация ***@NamedNativeQuery***.
5. **Запросы к хранимым процедурам** — новый API для вызова хранимых процедур, введенный в JPA 2.1 и обеспечивающий более высокую производительность JPQL-запросов, благодаря предварительной компиляции хранимой процедуры в базе данных.

Criteria API — типобезопасный вариант JPQL-запросов.

Мы ограничимся кратким рассмотрением только одного типа JPQL-запросов, известного как ***Criteria API***. Поскольку он использует только методы и аргументы языка Java, которые проходят синтаксический анализ во время компиляции, то такой подход называется ***типобезопасным***.

Каждый Criteria-запрос к базе данных формируется и осуществляется за несколько этапов. Рассмотрим эти этапы, используя для наглядности в качестве сущности класс ***Letter***.

Этап 1. Используя объект ***em*** менеджера ***EntityManager*** создаем объект ***builder*** класса ***CriteriaBuilder***:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
```

Этап 2. Создаем объект ***criteria***, используя один из многочисленных методов объекта ***builder***. В зависимости от прикладного назначения запроса, обычно используются следующие методы:

```
CriteriaQuery<Letter> criteria =
        builder.createQuery(Letter.class);

CriteriaDelete<Letter> criteria =
        builder.createCriteriaDelete(Letter.class);

CriteriaUpdate<Letter> criteria =
```

```
builder.createCriteriaUpdate(Letter.class);
```

Этап 3. Создаем объект ***root*** типа ***Root<T>***, который соответствует объектному представлению класса в JPQL и который можно использовать для обозначения атрибутов класса:

```
Root<Letter> root = criteria.from(Letter.class);
```

Например, указание на атрибут ***date*** сущности ***Letter*** будет выглядеть как: ***root.get("date")***.

Этап 4. Связываем объект ***criteria*** с объектом ***root***:

```
criteria.select(root);
```

Этап 5. Добавляем объекту ***criteria*** различные ограничения с помощью собственных методов, таких как: ***where(...)***, ***orderBy(...)***, ***groupBy(...)*** и ***having(...)***.

Этап 6. Формируем объект запроса ***query*** с помощью метода объекта ***em*** типа ***EntityManager***:

```
TypedQuery<Letter> query = em.createQuery(criteria);
```

Этап 7. Получаем результат, в зависимости от прикладного назначения сформированного запроса:

```
List<Letter> list = query.getResultList();

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

Этап 8. Обрабатываем полученный результат.

Среды разработки типа Eclipse EE достаточно качественно обеспечиват сопровождение первых семи этапов формирования Criteria-запросов.

Первоначально может показаться, что использование Criteria API — гораздо сложнее динамических, именованных или «*родных*» запросов. Это — действительно так, когда запросы не содержат сложных условий, но если получить определенный навык, то результат оправдает себя.

3.3.2 Реализация EJB-компонента с JTA-типом транзакций

Stateless EJB-контейнер способен инкапсулировать в себя объект менеджера сущностей ***EntityManager*** и самостоятельно управлять транзакциями ***JTA***-типа.

В предыдущем подразделе использовалась ***среда управляемая приложением***, что соответствовало единице сохранения «***lab4-unit1***». Если приложение разрабатывается как сервлет или EJB-контейнер, то возможно использование ***среды управляемой контейнером***.

Приложение, функционирующее в среде управляемой контейнером, может сразу инкапсулировать менеджер сущностей ***EntityManager*** с помошью аннотации ***@PersistenceContext (name="имя юнита"***).

В нашей учебной среде за связь с технологией JPA, управляемой транзакциями типа ***transaction-type="JTA"***, отвечает юнит «***lab4-unit2***», описанный в файле ***persistence.xml***. Проведем демонстрацию работы Stateless EJB-контейнера, реализовав EJB-компоненту с именем ***Lets2***, которая в свою очередь реализует интерфейсы ***LocalLetter*** и ***RemoteLetter***.

Фактически реализация EJB-компоненты с именем ***Lets2*** — это полная реализация EJB-компоненты ***Lets1***, с учетом новых технологий запросов к учебной базе данных: JTA-тип транзакций и Criteria-запросы. Результат такой реализации представлен на листинге 3.24.

Листинг 3.24 — Исходный текст EJB-компоненты Lets2.java проекта lab4

```
package rsos.lab4;
import java.util.List;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

@Stateless
@LocalBean
public class Lets2 implements LocalLetter, RemoteLetter
{
	@PersistenceContext(name = "lab4-unit2")
	private EntityManager em;

	// Получение списка записей таблицы t_letter
	public List<Letter> getList()
	{
```

```
        // Этап 1. Создаем объект builder
        CriteriaBuilder builder =
              em.getCriteriaBuilder();

                    // Этап 2. Создаем объект criteria
        CriteriaQuery<Letter> criteria =
                    builder.createQuery(Letter.class);

        // Этап 3. Создаем объект root
        Root<Letter> root =
              criteria.from(Letter.class);

        // Этап 4. Преобразовываем объект criteria,
        // включая использование объекта root
        criteria.select(root);

        // Этап 5. Добавляем сортировку
        criteria.orderBy(builder.desc(root.get("date")));

        // Этап 6. Формируем запрос в виде объекта query
        TypedQuery<Letter> query = em.createQuery(criteria);

        // Этап 7. Получаем результат
        List<Letter> list =
              query.getResultList();

        // Этап 8. Обрабатываем результат
        return list;
  }

  // Получение объекта по ключу
  public Letter getLetter(int id) {
        Letter l =
              em.find(Letter.class, new Integer(id));
        return l;
   }

   // Добавить объект в базу данных
   public void addLetter(Letter letter) {
        em.persist(letter);
   }

   // Удалить объект из базы данных по ключу
   public void deleteLetter(int id) {
        Letter letter =
              em.find(Letter.class, new Integer(id));
              if(letter == null)
                    return;
        em.remove(letter);
   }

   // Модифицировать объект в базе данных
   public void modLetter(Letter letter) {
        if(letter == null)
                    return;
        Integer Id = letter.getId();
```

```
            Letter l =
                  em.find(Letter.class, Id);
            if(l == null)
                  return;

            //System.out.println("Letters: модифицирую №" + Id);
            l.setDate(letter.getDate());
            l.setName(letter.getName());
            l.setText(letter.getText());
      }
}
```

При анализе данного листинга следует обратить внимание на способ инкапсуляции объекта менеджера сущностей ***em*** со ссылкой на юнит «***lab4-unit2***» (выделено серым цветом). В результате не нужно использовать ни фаб-рику менеджеров ни менеджер транзакций.

Самым сложным в реализации является метод ***getList()***, но он полностью комментирован по этапам реализации Criteria-запросов, описанных в предыдущем пункте. Дабавлена также сортировка по атрибуту ***date***.

Следует обратить внимание и на метод ***modLetter(...)***, модифицирующий уже существующий объект базы данных:

1) ***сначала***, из представленного в качестве аргумента объекта извлекается ключ модифицеруемой записи (объекта);
2) ***затем***, модифицируемый объект извлекается из базы данных;
3) ***наконец***, переустанавливаются атрибуты извлеченного объекта и — все; далее, менеджер сущностей сам обеспечит модификацию объекта.

3.3.3 Реализация JPA-сервлета

Для получения полноценного демонстрационного примера, необходимо реализовать соответсвующий HTTP-сервлет и JSP-страницу.

Проведем реализацию сервлета ***JpaServlet2***, инкапсулирующего EJB-компоненту ***Lets2*** и обеспечивающего полную функциональность учебной задачи ***Letters*** (***Письма***), ранее заявленной в пункте 3.1.1. Для этого, сначала реализуем новую JSP-страницу в файле ***jpa_test2.jsp***, исходный текст которой представлен на листинге 3.25.

Листинг 3.25 — Исходный текст JSP-страницы jpa_test2.jsp проекта lab4

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
```

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

<jsp:useBean id="letter2" class="rsos.lab4.Letter" scope="page" />

<html>
<head>
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
       <title>lab4</title>
</head>
<body>
       <hr>
            <b>Тестовая страница проекта lab4 (jpa_test2.jsp)</b>
       <hr>
       <table id="table1" cellspacing="0" cellpadding="5" border="0" >
        <thead><tr>

            <th align="left" >№</th>
            <th align="left" >Дата</th>
            <th align="left" >Пользователь</th>
            <th align="left" >Сообщение</th>

        </tr></thead>
        <tbody>
        <c:forEach items="${lists2}" var="letter2">
            <tr>
            <td width="8">
            <c:out value="${letter2.id}" default="*"                 /></td>
            <td width="8">
            <c:out value="${letter2.dateString}" default="Нет даты"  /></td>
            <td width="8">
            <c:out value="${letter2.name}" default="Нет пользователя"/></td>
            <td>
            <c:out value="${letter2.text}" default="Нет текста..."   /></td>
            </tr>
         </c:forEach>

        </tbody>
    </table>
         <hr>
         <form action="JpaServlet2" method="post" accept-charset="UTF-8">
         <table id="table2">
           <tr>
                 <td>Номер сообщения: </td>
                 <td><input type="text" size="6" name="id"
                       value="${id}" /></td>
           </tr>
           <tr>
                 <td>Имя пользователя: </td>
                 <td><input type="text" size="40" name="user"
                       value="${user}" /></td>
           </tr>
           <tr>
                 <td>Новый текст: </td>
                 <td><textarea rows="5" cols="40" name="text">
                       <%= request.getAttribute("text") %> </textarea></td>
           </tr>
```

```
            </table>
        <hr>
        Состояние запроса (<b><i><%= request.getAttribute("action")%></i></b>):
                           <b><i><%= request.getAttribute("state") %></i></b>
        <hr>
        <table id="table3">
           <tr>
              <td>
              <input type="radio" name="action" value="list" checked="checked">
                           Обновить
              </td>
              <td><input type="submit" value="Отправить запрос" />
                    </td>
           </tr>
           <tr>
                  <td>
                    <input type="radio" name="action" value="add">
                          Добавить<br>
                    <input type="radio" name="action" value="get">
                          Прочитать по номеру<br>
                    <input type="radio" name="action" value="delete">
                          Удалить по номеру<br>
                    <input type="radio" name="action" value="mod">
                          Модифицировать по номеру<br>
                    </td>
           </tr>
        </table>
        </form>
        <hr>

</body>
</html>
```

Наиболее важные места представленного листинга, которые должны быть учтены в сервлете ***JpaServlet2***, выделены серым фоном.

Что касается самого сервлета ***JpaServlet2***, то его общая структура совпадает с предыдущей реализацией. Отличия состоят в инкасуляции нового EJB-компонента ***Lets2*** и реализации всех запросов, что показано на листинге 3.26. Измененные места также выделены серым фоном.

Листинг 3.26 — Исходный текст сервлета JpaServlet2 проекта lab4

```
package rsos.lab4;
import java.io.IOException;
import java.util.Date;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class JpaServlet2
```

```
 */
@WebServlet("/JpaServlet2")
public class JpaServlet2 extends HttpServlet
{
        private static final long serialVersionUID = 1L;

        // Инкапсуляция EJB-компонента типа Lets2
        @EJB
        private Lets2               lets2;      // EJB-компонент

        /**
          * Конструктор
          */
          public JpaServlet2() { super(); }

          /**
           * Метод doGet(...)
           */
          protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                              throws ServletException, IOException
          {
                  // Переходим к методу doPost()
                  doPost(request, response);
          }
          /**
           * Метод doPost(...)
           */
          protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
                        throws ServletException, IOException
          {
                  /**
                   * Явная установка кодировок объектов запроса и ответа.
                   * Стандартная установка контекста ответа.
                   */
                              response.setCharacterEncoding("UTF-8");
                  response.setContentType("text/html");
                  request.setCharacterEncoding("UTF-8");

                  // Начальные значения
                  String   state  = "<b>Нормально...</b>";
                  String   action = request.getParameter("action");
                  String   sId    = request.getParameter("id");
                  Integer  Id     = ToInt.get(sId);
                  String   user   = request.getParameter("user");
                  String   text   = request.getParameter("text");

                        if(action == null) action   = "list";
                  if(user   == null) user     = "asu";
                  if(user.trim().length() == 0) user = "asu";
                  if(text   == null) text     = "";

                  request.setAttribute("state", state);

                  // Действия
                  if("add".equals(action)) {
```

```
                        lets2.addLetter(new Letter(new Date(), user, text));
                        text = "";
                  }

                  if("get".equals(action)) {
                        if(Id == null) {
                              request.setAttribute("state",
                              "<b>Ошибочный идектификатор сообщения...</b>");
                              text = "";
                  } else {
                        Letter letter =
                              lets2.getLetter(Id.intValue());
                        if(letter == null) {
                              request.setAttribute("state",
                                    "<b>Нет такого сообщения...</b>");
                              text = "";
                        } else {
                              user = letter.getName();
                              text = letter.getText();
                        }
                  }
            }

            if("delete".equals(action)) {
                  if(Id == null) {
                        request.setAttribute("state",
                        "<b>Ошибочный идектификатор сообщения...</b>");
                        text = "";
                  } else {
                        lets2.deleteLetter(Id.intValue());
                        text = "";
                  }
            }

            if("mod".equals(action)) {
                  if(Id == null) {
                        request.setAttribute("state",
                        "<b>Ошибочный идектификатор сообщения...</b>");
                  } else {
                        Letter l =
                              new Letter(new Date(), user, text);
                        l.setId(Id.intValue());
                        lets2.modLetter(l);
                  }
            }

            request.setAttribute("lists2",  lets2.getList());

            // Установка атрибутов
            request.setAttribute("action",  action);
            request.setAttribute("id",      sId);
            request.setAttribute("user",    user);
            request.setAttribute("text",    text.trim());
            /**
             * Стандартное подключение ресурса JSP-страницы
             */
            request.getRequestDispatcher("/WEB-INF/jpa_test2.jsp")
```

```
                .forward(request, response);
        }
}
```

Результат запуска сервлета ***JpaServlet2*** показан на рисунке 3.7. Хорошо видно, что этот результат полностью совпадает с запуском ***JpaServlet1***, представленным ранее на рисунке 3.6.

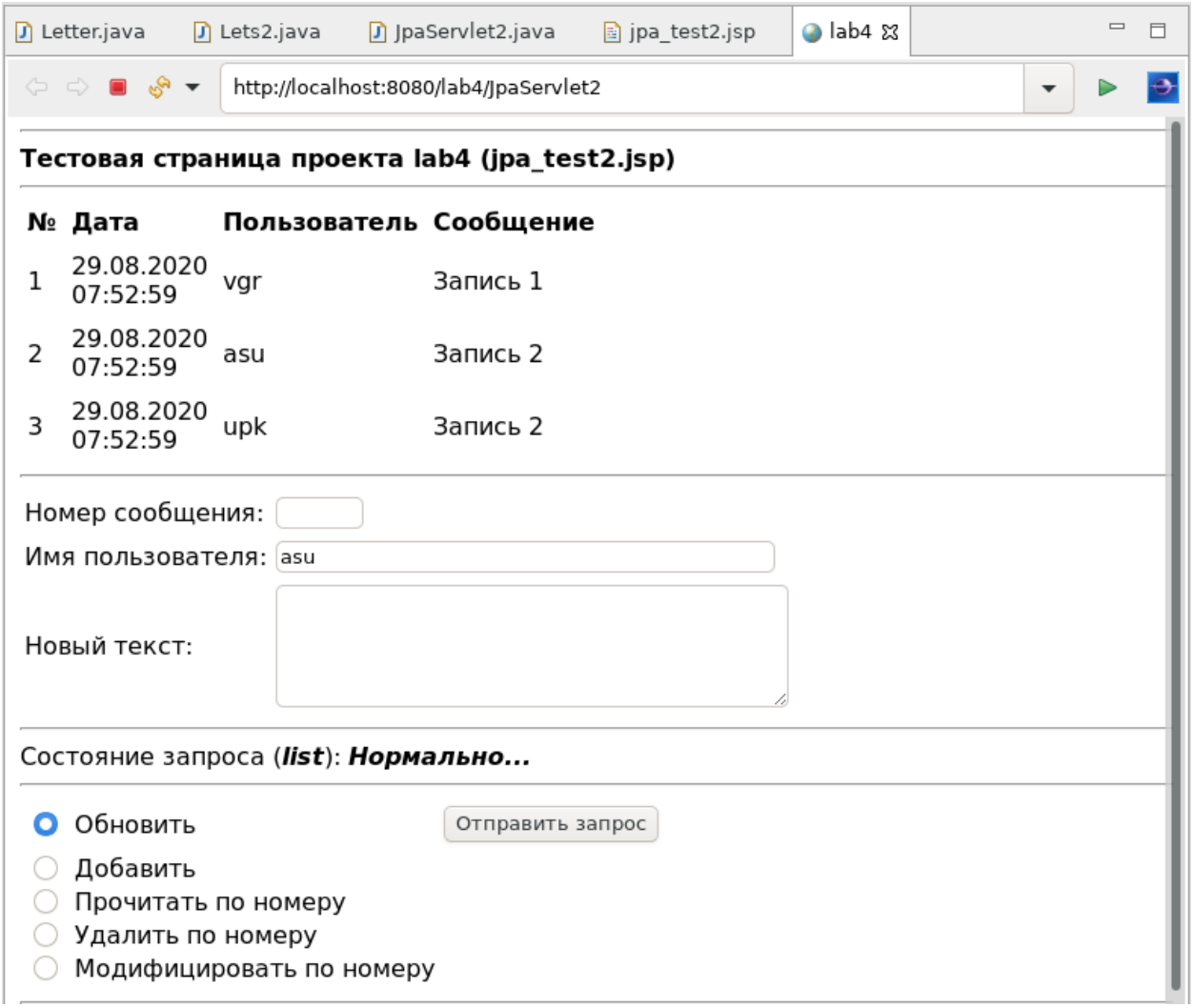

Рисунок 3.7 — Результат запуска сервлета JpaServlet2

Продолжая тестирование, выполним два действия:

1) ***добавим*** текст: «*Тест сервлета JpaServlet2.*»;
2) ***удалим***, например, запись №2.

В результате мы получим измененный список содержимого базы данных, показанный на рисунке 3.8. Дополнительно, нужно проверить результат изменения базы данных запустив SQL-сценарий ***select_lab4db.sql***.

Обратите также внимание, что список записей отсортирован по убыванию

даты внесения изменений, как это сделано — смотрите метод ***getList()*** EJB-компонента ***Lets2***, представленного ранее на листинге 3.24.

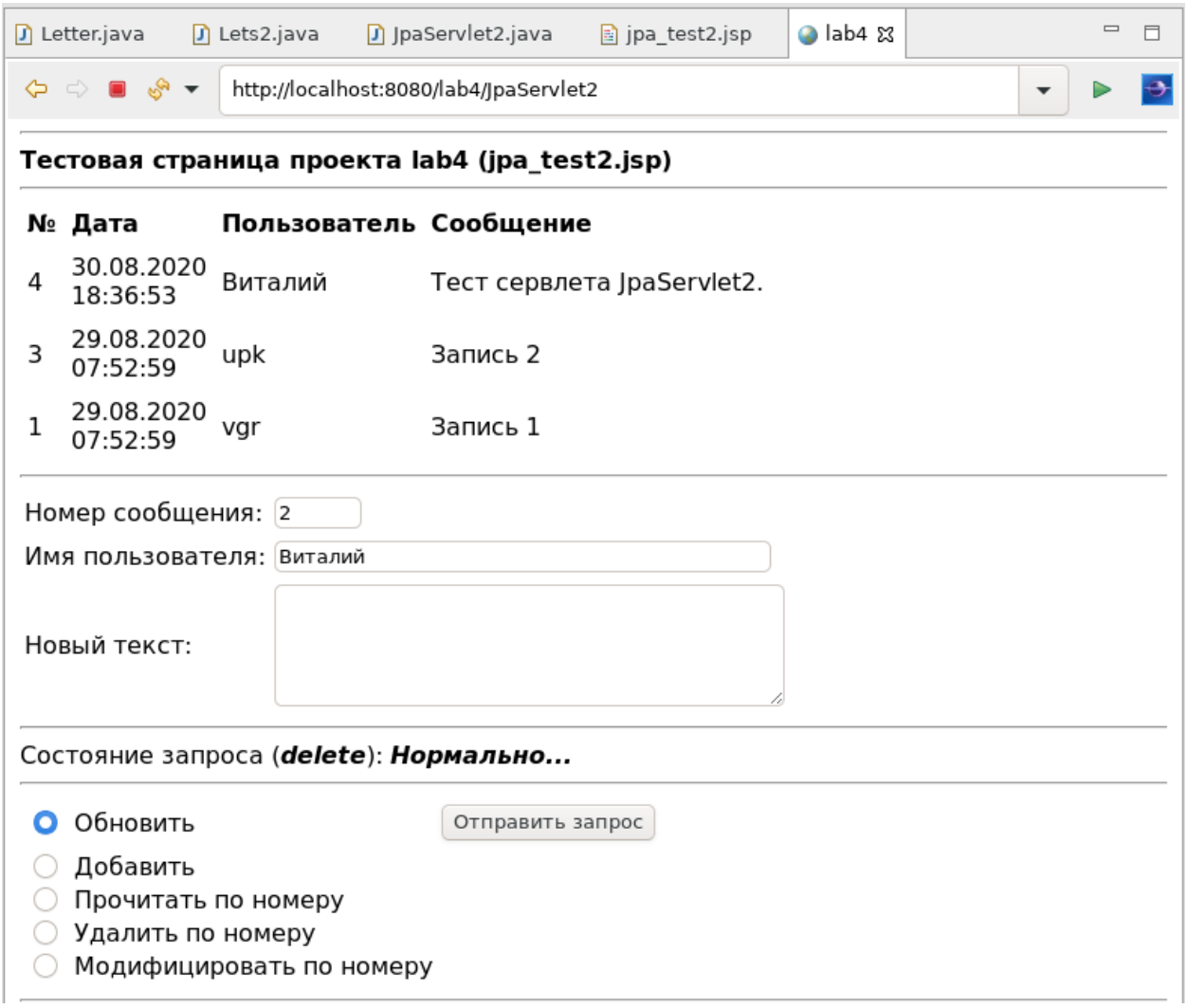

Рисунок 3.8 — Результат работы сервлета, после проведенных изменений

В завершение данной главы отметим, что мы с вами кратко изучили теорию и практику создания EJB-компонент программной платформы Java EE, которые обычно всегда работают с теми или иными базами данных. В плане общего контекста изучаемой дисциплины, EJB-компоненты являются важной частью распределнных сервис-ориентированных систем и рассматриваются в них как распределенные приложения.

Хотя все практические результаты получены в среде Web-серверов, это не умаляет результаты полученных студентом знаний. В дальнейшем, теоретическая и практическая части данной главы будут использованы в учебном материале последующих глав.

Вопросы для самопроверки

1. Что означает абревиатура JPA и где она применяется?
2. Чем отличаются технологии JPA и JTA?
3. В чем состоит проблема использования языка SQL в распределенных сервис-ориентированных системах?
4. Чем отличаются EJB-компоненты от EJB Light-компонент и какими аннотациями они обозначаются ?
5. На какие три типа разделяются EJB-компоненты и какими аннотациями они обозначаются?
6. Что означает понятие сессионного EJB-компонента?
7. Какие интерфейсы бывают у EJB-компонента и какими аннотациями они обозначаются?
8. Что такое - EJB-компонент без обеспечения интерфейса и какой аннотацией он обозначаются?
9. Какие дескрипторы файлов необходимы серверу приложений Apache TomEE для использования технологии JPA?
10. Какие типы транзакций используют провайдеры технологии JPA?
11. Что такое — сущности и какой аннотацией они обозначаются?
12. Что такое — объектно-реляционное отображение?
13. Что такое — менеджер сущностей?
14. Что означает сокращение JPQL и где оно используется?
15. Что такое — Stateless EJB-контейнер и каковы его возможности?
16. Чем отличаются JTA и не-JTA типы транзакций?
17. Что такое — менеджер транзакций? Где и как он используется?
18. Что такое — транзакции управляемые приложением?
19. Что такое — транзакции управляемые контейнером?
20. Назовите пять типов JPQL-запросов?

4 Тема 4. Обработка документов XML и JSON

Язык XML составляет основу представления данных современных сервис-ориентированных технологий.

В первой главе пособия и далее мы много говорили о форматах представления информации на языке XML. Прежде всего это касалось различных дескрипторов развертывания: ***web.xml***, ***context.xml***, ***beans.xml***, ***resources.xml***, ***persistence.xml*** и других. Мы пытались минимизировать их использование, заменяя дескрипторы развертывания соответствующими аннотациями. Тем не менее, формат представления XML используется и для многих других целей, например, представления документов, формировании запросов Web-служб и вообще — для хранения информации, состоящей из следующих понятий:

1. **Симолы Unicode** — XML-документ является текстовой строкой, способной использовать любые доступные символы Unicode.
2. **Разметка и содержимое** — выделение в тексте XML-документа тегов, ограниченных угловыми скобками, а все остальное является содержимым.
3. **Теги** — значимые элементы Языка XML, которые подразделяются на открывающие, закрывающие и пустые.
4. **Элемент** — начинается с открывающего тега и заканчивается закрывающим тегом или состоит только из пустого тега.
5. **Атрибут** — представляет собой пару «имя/значение», которая располагается в открывающем или закрывающем теге.
6. **Объявление XML** — <?xml version="1.0" encoding="UTF-8" ?>

Учебная цель данной главы — изучение технологий платформы Java EE для преобразования объектов Java-классов в формат представления XML и обратно.

В целом учебный материал представлен в виде двух подразделов:

1) подраздел 4.1 посвящен технологии JAXB, которая напрямую реализует поставленную учебную цель;
2) подраздел 4.2 описывает технологию JXON, которая предназначена для повышения эффективности представления данных в запросах у Web-сервисам, рассмотренным в последующих двух главах.

Таким образом, данная глава завершает изучение инструментальных средств, необходимых для проектирования и реализации Web-сервисов.

4.1 Технология JAXB

Язык Java имеет множество инструменов для обработки документов в формате XML. Все они объединяются под общим название ***JAXP*** или ***Java Architecture for XML Processing***. Нам, в данной дисциплине, интересна технология связывания объектов JAVA-классов с представлением в формате языка XML, исвестная как JAXB.

JAXB (*Java Architecture for XML Binding*) — технология, позволяющая ставить в соответствие JAVA-классы и XML-представления, предоставляет две основные возможности:

а) ***маршалинг*** JAVA-объектов в формат документа XML;

б) ***демаршалинг*** из документа XML обратно в JAVA-объект.

Рассмотрим сначала инструментальную часть языка Java, а затем — проведем демонстрацию этого инструментария конкретным примером.

4.1.1 Программное обеспечение технологии JAXB

Маршалинг (*Marshaling*) — упорядочивание или — процесс преобразования информации (данных или двоичного представления объекта), хранящейся в оперативной памяти, в формат, пригодный для хранения или передачи.

Демаршалинг (*Unmarshaling*) — обратная операция распаковки из стандартного формата в форму, приемлемую для принимающего процесса, которая сходна с операцией десериализации.

JAXB API — инструментарий языка Java, определенный в пакете ***javax.xml.bind*** и предоставляющий набор интерфейсов и классов для создания XML-документов и генерации классов Java.

Общий набор инструментов JAXB API представлен в таблице 4.1.

Центральное место в рассматриваемой технологии JAXB API занимает класс ***javax.xml.bind.JAXBContext***, который и управляет связыванием между XML-документами и объектами Java.

Класс ***JAXBContext***, с помощью метода ***newInstance(<класс>)***, предоставляет программисту объект контекста связывания, на основе которого формируются другие объекты преобразований. Например, для получения контекста класса ***Letter*** (см. листинг 3.1, стр. 134), необходимо выполнить:

```
JAXBContext context =
        JAXBContext.newInstance(Letter.class);
```

Таблица 4.1 — Пакеты технологии JAXB [17]

Пакет	***Описание пакета***
javax.xml.bind	Фреймворк связывания среды выполнения, имеющий возможность выполнять операции маршалинга, демаршалинга и проверки.
javax.xml.bind.annotation	Аннотации для настройки преобразований между программой Java и XML-данными.
javax.xml.bind.annotation.adapters	Классы-адаптеры JAXB.
javax.xml.bind.attachment	Выполнение маршалинга для оптимизации хранения двоичных данных и демаршалинг корня документа, содержащего форматы двоичных данных.
javax.xml.bind.helpers	Частичные стандартные реализации некоторых интерфейсов ***javax.xml.binding***.
javax.xml.bind.util	Набор вспомогательных классов.

На основе созданного объекта ***context*** можно создать еще два объекта типа ***Marshaller*** и ***Unmarshaller***:

```
Marshaller marsh =
        context.createMarshaller();

Unmarshaller unmarsh =
        context.createUnmarshaller();
```

Дополнительно, объекту ***marsh*** можно установить свойство, требующее обеспечить вывод в формат XML в удобном читаемом виде, а не просто -вывод в виде текстовой строки:

```
marsh.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                  Boolean.TRUE);
```

Теперь, используя методы объектов ***marsh*** и ***unmarsh***, можно обеспечить преобразование объекта класса ***Letter*** в XML формат и обратно:

```
marsh.marshal(Letter letter, <объект вывода>);

Letter letter =
        (Letter) unmarsh.unmarshal(<объект ввода>);
```

Обратите внимание, что методы ***marshal(...)*** и ***unmarshal(...)*** требуют указания объектов ввода и вывода.

Для вывода на терминал можно использовать объект ***System.out***, а для вывода в файл — объект класса ***File***.

Для чтения и восстановления объекта из XML-файла лучше использовать класс ***FileReader***.

В общем случае, объектами чтения и записи являются потоки ввода/вывода ***InputStream*** и ***OutputStream***.

4.1.2 Аннотации для связывания объектов Java

Правильное отображение JAVA-объекта в XML-представление требует использования аннотаций.

Полное описание технологии JAXB — достаточно объемно. Его можно найти по ссылке источника [26]. Наиболее важные аннотации, регулирующие правильное отображение объектов Java в XML-представление и достаточные для демонстрации простейших примеров, выборочно представлены в таблице 4.2.

Таблица 4.2 — Наиболее важные аннотации технологии JAXB [17]

Аннотация	***Описание***
@XmlRootElement(name)	Обязательная аннотация, необходимая любому классу для связывания в качестве корневого элемента XML. Ставится возле класса.
@XmlType(name, propOrder)	Аннотирует класс как комплексный тип в схеме XML. Ставится возле класса. Позволяет задавать имя тега и порядок отображения элементов.
@XmlElement(name)	Ставится около поля. Поле будет представлено в XML-элементом. Позволяет задать имя для тэга.
@XmlAttribute(name)	Ставится около поля и оно будет представлено в XML-атрибутом. Позволяет задать имя для атрибута.
@XmlElementWrapper(name, nillable = true)	Ставится около поля и позволяет задать обрамляющий тег для группы элементов. Позволяет задать имя для тэга.
@XmlJavaTypeAdapter(...)	Ставится около поля и позволяет задать класс, который будет преобразовывать данные поля в строку.
@XmlTransient	Ставится возле поля и информирует JAXB, что не нужно связывать атрибут.

Обратите внимание, что почти все аннотации имеют атрибут ***name***, который позволяет изменить отображаемое имя поля или класса. Если этот атрибут не указан, то имя поля или класса не изменяется. Кроме того, если переменная класса имеет спецификатор доступа ***private***, то аннотация ставится передсоответствующим публичным методом ***get*(...)***.

Если никакой атрибут перед полем не ставится, то имя поля не меняется и считается, что оно отображается как элемент.

Важно также отметить, что при отображении поля могут следовать не в том порядке, как они заданы в описании класса Java. Чтобы задать необходимый порядок отображения атрибутов, используется параметр ***propOrder*** аннотации ***@XmlType***.

Для примера, рассмотрим класс ***Letter***, представленный ранее на листинге 3.1 (стр. 134), в котором поля заданы в следующем порядке: ***id***, ***date***, ***name*** и ***text***.

Допустим мы хотим, чтобы объект класса ***Letter*** отображался в XML-файл под корневым тегом <***letter-object***> и содержал приватную переменную ***id*** в качестве атрибута, а остальные приватные переменные ***date***, ***name*** и ***text*** — как элементы. Тогда исходный текст класса ***Letter*** должен быть аннотирован, как показано на листинге 4.1.

Листинг 4.1 — Аннотация класса Letter.java для отображения в XML

```
package rsos.lab5;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder={"date", "name", "text"})
@XmlRootElement(name="letter-object")
public class Letter implements Serializable
{
    // Идентификатор сериализации
    private static final long serialVersionUID = 1L;

    // Поля данных класса Letter
    private int id;
          private Date date;
    private String name;
    private String text;

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
    public Letter() {
    }

    public Letter(Date date, String name, String text) {
        this.date = date;
        this.name = name;
        this.text = text;
    }

    // Геттеры и сеттеры POJO-класса
```

```
    @XmlAttribute
    public int getId() {
        return id;
    }

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

    @XmlElement
    public Date getDate() {
        return this.date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    @XmlElement
    public String getName() {
        return name;
    }

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

    @XmlElement
    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    // Дополнительные функции форматирования
    public String toString() {
      return Integer.toString(id) + " " + date.toString() + " "
                  + name + " " + text;
    }

    public String getDateString() {
      SimpleDateFormat sdf =
                  new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      return sdf.format(this.date);
    }
}
```

В целом, заканчивая теоретическую часть изучения технологии JAXB, отметим, ее возможности — гораздо шире тех, что рассмотрены здесь. Для наших целей полученной информации — вполне достаточно, чтобы обеспечить теоретические и практические потребности изучения последующих тем. Студентам, желающим более подробно изучить технологию, рекомендуем воспользовантся документацией [26] или иным подходящим источником.

4.1.3 Преобразование объекта Java в документ XML

Проведем демонстрацию теоретических рассуждений, изложенных в предыдущих двух пунктах, посредством реализации нескольких тестовых приложений.

Для этого, в инструментальной среде Eclipse создадим обычный Java-проект с именем ***lab5***. Откроем в нем новый класс с именем ***Letter*** и перенесем в него содержимое исходного текста, который представлен ранее на листинге 4.1. Далее, создадим новый Java-класс с именем ***Test1*** и реализуем в нем алгоритм отображения объекта класса ***Letter*** в XML-файл с абсолюным имененем ***/home/upk/lab5.Letter.xml***.

Полная реализация класса ***Test1*** со всеми необходимыми комментариями приведена на листинге 4.2.

Листинг 4.2 — Исходный текст класса Test1 проекта lab5

```
package rsos.lab5;

import java.io.File;
import java.util.Date;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class Test1 {

	public static void main(String[] args) throws JAXBException
	{
		// Создаем объект класса Letter
		Letter let1 =
			new Letter(new Date(), "upk",
					"Сообщение для lab5: приложение Test1");
		let1.setId(20);

		// Печатаем содержимое объекта let1 на консоль, используя
		// переопределенный метод toString() класса Letter
		System.out.println("Объект let1=[" + let1 + "]\n");

		// Создаем контекст технологии JAXB для класса Letter
		JAXBContext context =
				JAXBContext.newInstance(Letter.class);

		// Создаем объект маршалинга
		Marshaller marsh =
				context.createMarshaller();

		// Добавляем свойство форматированного вывода
		marsh.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
				Boolean.TRUE);
```

```
            // Выводим отображение объекта let1 на терминал
            marsh.marshal(let1, System.out);

            // Создаем объект вывода в файл: /home/upk/lab5.Letter.xml
            File file =
                  new File(System.getenv("HOME") + "/lab5.Letter.xml");

            // Выводим отображение объекта let1 в файл
            marsh.marshal(let1, file);
      }
}
```

Первый тест проведем, запустив класс ***Test1*** на выполнение, причем в отображаемом классе ***Letter*** оставим только одну обязательную аннотацию ***@XmlRootElement*** (без параметра ***name***). Убедимся, что отображение исследуемого класса записалось в файл ***lab5.Letter.xml***, как это показано на рисунке 4.1.

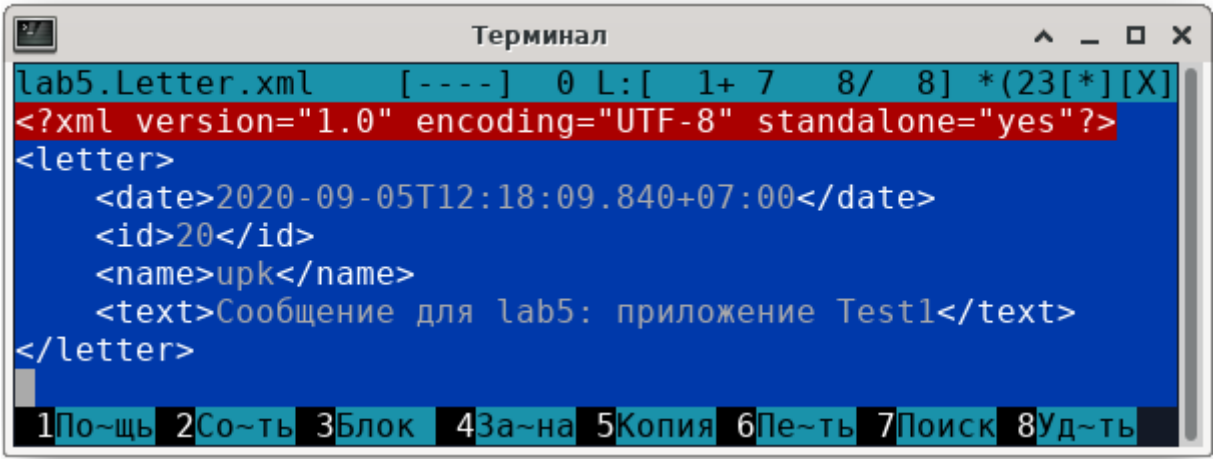

Рисунок 4.1 — Прямое отображение класса Letter в файл lab5.Letter.xml

Обратите внимание, что все переменные класса ***Letter*** отобразились в файл со своими именами. Причем:

а) имя класса отобразилось в корневом теге и со строчной буквы;

б) переменные класса — ***id*** и ***date*** правильно отобразились тегами со своими именами, но поменялись местами.

Второй тест проведем, восстановив все аннотации класса ***Letter*** так, как они представлены на листинге 4.1.

Результат запуска второго теста представлен на рисунке 4.2, в виде изображения консоли инструментальной среды Eclipse.

Первой строкой на консоль выведено представление объекта ***let1*** с помощью метода ***toString()*** класса ***Letter***. Заметим, что этот метод является стандартным для системного вывода языка Java, а программисты используют его переопределения, когда им нужно контролировать содержимое сложных объектов.

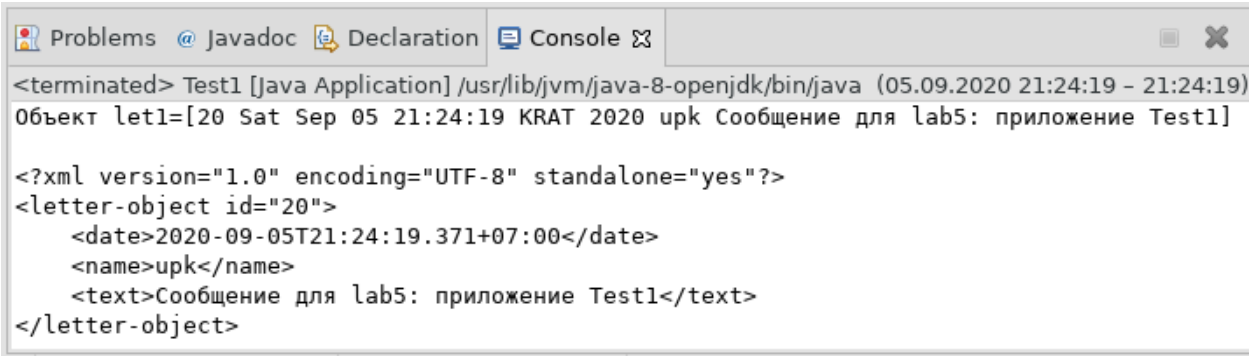

Рисунок 4.2 — Полностью аннотированное отображение класса Letter

Далее, через пустую строку, выведено отображение объекта ***let1*** для варианта полностью аннотированного класса ***Letter***. Здесь можно отметить, что:

а) изменилось название корневого тега на <***letter-object***>;

б) переменная класса ***id*** представлена как атирибут;

в) остальные переменные класса представлены как элементы своими тегами и в том порядке, как это указано в аннотации ***@XmlType(...)***.

Третий тест проведем с помощью запуска на выполнение класса ***Test2***, исходный текст которого приведен на листинге 4.3 и который производит обратное отображение XML-файла ***lab5.Letter.xml*** в объект ***let2*** класса ***Letter***.

Листинг 4.3 — Исходный текст класса Test2 проекта lab5

```
package rsos.lab5;

import java.io.FileNotFoundException;
import java.io.FileReader;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

public class Test2 {

	public static void main(String[] args)
			throws JAXBException, FileNotFoundException
	{
		// Создаем контекст технологии JAXB для класса Letter
		JAXBContext context =
				JAXBContext.newInstance(Letter.class);

		// Создаем объект демаршалинга
		Unmarshaller unmarsh =
				context.createUnmarshaller();

		// Создаем объект вода из файла: /home/upk/lab5.Letter.xml
		FileReader fileReader =
			new FileReader(System.getenv("HOME") + "/lab5.Letter.xml");
```

```
            // Водим отображение объекта let1 из файла:
            // /home/upk/lab5.Letter.xml
            Letter let2 =
                  (Letter) unmarsh.unmarshal(fileReader);

            // Печатаем содержимое объекта let2 на консоль, используя
            // переопределенный метод toString() класса Letter
            System.out.println("Объект let2=[" + let2 + "]\n");
      }
}
```

Запустив класс ***Test2*** на выполнение, мы получим результат, показанный на рисунке 4.3.

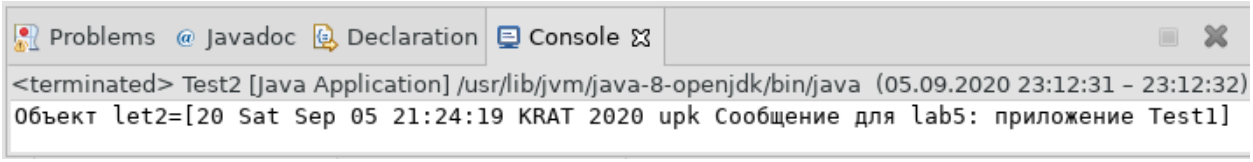

Рисунок 4.3 — Обратное отображение класса Letter в объект let2

Хорошо видно, что объект ***let2*** класса ***Letter*** полностью и точно востановлен из сохраненного отображения ***lab5.Letter.xml***.

Обычно, данные в приложениях передаются списками.

Рассмотрим более сложное отображение объектов JAVA-классов. Для этого, рассмотрим класс ***ListLetters***, содержащий две переменные:

1) ***name*** — имя или название списка;
2) ***list*** — список объектов класса ***Letter*** (возможно пустой).

Исходный и полностью аннотированный текст сериализованного класса ***ListLetters*** представим листингом 4.4.

Листинг 4.4 — Исходный текст класса ListLetters проекта lab5

```
package rsos.lab5;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder={"name", "list"})
@XmlRootElement
```

```
public class ListLetters implements Serializable
{
    private static final long serialVersionUID = 1L;

    // Приватные переменные класса
    private String name;
    private List<Letter> list = new ArrayList<>();

    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
    public ListLetters() {
    }

    public ListLetters(String name) {
        this.name = name;
    }

    @XmlElement
    public String getName() {
        return name;
    }

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

    @XmlElementWrapper(name="list-letters", nillable = true)
    public List<Letter> getList() {
        return list;
    }

    public void setList(List<Letter> list) {
        this.list = list;
    }

    // Дополнительная функция форматирования
    public String toString() {
      String s0 = "ListLetters:name=" + name;
      for(Letter let : list)
      {
            s0 += "\n" + let.toString();
      }
      return s0;
    }
}
```

Обратите внимание, что ***ListLetters*** представляет собой POJO-класс с двумя приватными переменными, поэтому аннотации выставляются перед публичными методами ***get*()***.

Добавилась новая аннотация ***@XmlElementWrapper(...)***, которая соотносится с переменной ***list*** вместо аннотации ***@XmlElement*** и вводит новое имя ***list-letters***, которое будет «*окружать*» (*wrapper*) список объектов типа ***Letter***.

У класса ***ListLetters*** имеется также собственный переопределенный метод

toString(), назначение которого — обеспечить системный вывод содержимого объекта на терминал (консоль). Этот метод не является обязательным, но очень полезен для наших учебных целей.

Теперь создадим тестовый класс ***Test3***, обеспечивающий создание объекта ***list*** типа ***ListLetters***, отображение этого объекта в XML-формат и запись полученного отображения в файл ***/home/upk/lab5.ListLetters.xml***.

Сам объект ***list*** будет иметь собственное имя и содержать список из двух объектов типа ***Letter***, как это показано на листинге 4.5.

Листинг 4.5 — Исходный текст класса Test3 проекта lab5

```
package rsos.lab5;
import java.io.File;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class Test3 {

	public static void main(String[] args) throws JAXBException
	{
		// Создаем список из двух объектов класса Letter
		List<Letter> lets = new ArrayList<>();

		// Первое сообщение
		Letter letter =
			new Letter(new Date(), "upk",
					"Сообщение №21 для lab5:Test3");
		letter.setId(21);
		lets.add(letter);

		// Второе сообщение
		letter =
			new Letter(new Date(), "asu",
				"Сообщение №22 для lab5:Test3");
		letter.setId(22);
		lets.add(letter);

		// Создаем объект класса ListLetters
		ListLetters list =
			new ListLetters("Список писем для приложения lab5:Test3");
		list.setList(lets);

		// Печатаем содержимое объекта list на консоль, используя
		// переопределенный метод toString() класса ListLetters
		System.out.println("Объект list\n----------\n" + list + "\n");

		// Создаем контекст технологии JAXB для класса ListLetters
		JAXBContext context =
			JAXBContext.newInstance(ListLetters.class);
```

```
            // Создаем объект маршалинга
            Marshaller marsh =
                        context.createMarshaller();

            // Добавляем свойство форматированного вывода
            marsh.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                              Boolean.TRUE);

            // Выводим отображение объекта list на терминал
            marsh.marshal(list, System.out);

            // Создаем объект вывода в файл: /home/upk/lab5.ListLetters.xml
            File file =
                  new File(System.getenv("HOME") + "/lab5.ListLetters.xml");

            // Выводим отображение объекта list в файл
            marsh.marshal(list, file);
        }
}
```

Общая структура класса ***Test3*** — полностью аналогична структуре класса ***Test1***. Отличия состоят только в подготовке исходных данных для отображения и выделены на листинге серым фоном.

Третий тест представлен на рисунке 4.4, а студенту предлагается самостоятельно реализовать обратное отображение в классе ***Test4***.

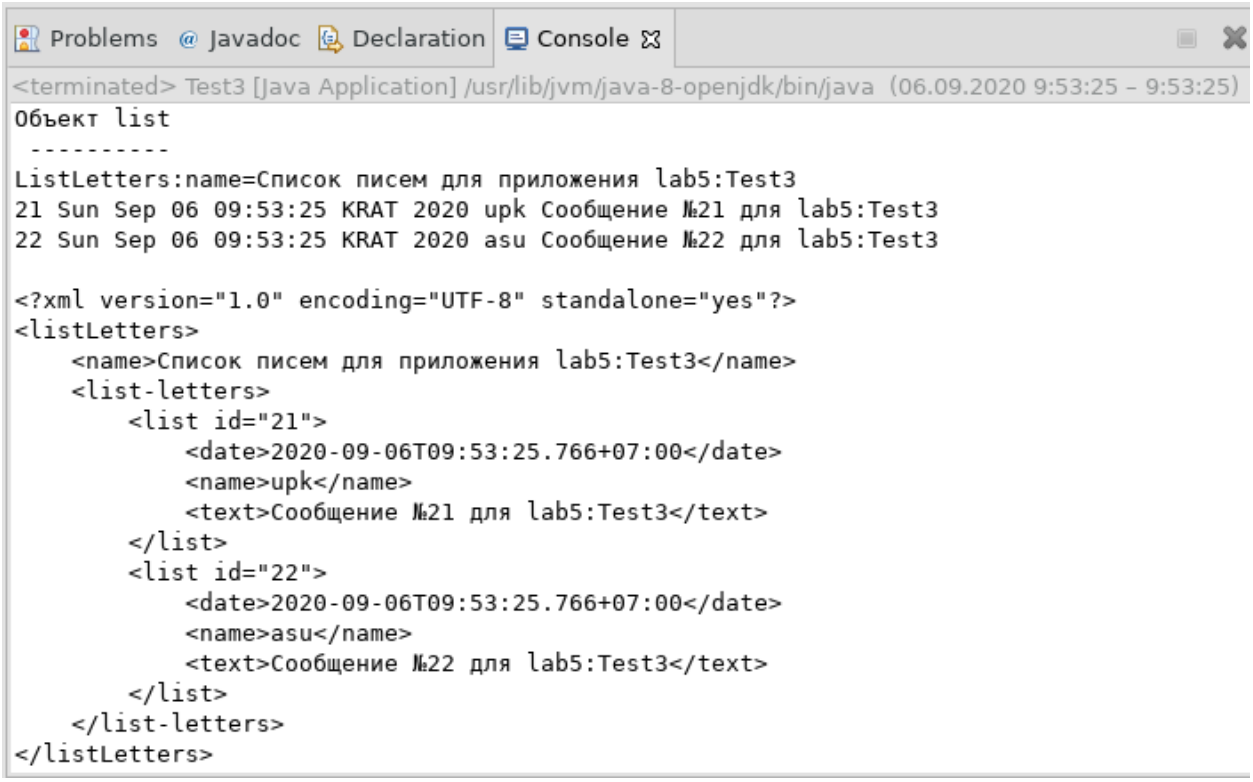

Рисунок 4.4 — Полностью аннотированное отображение класса ListLetters

4.2 Технология JSON

Частов качестве альтернативы XML-формату, при взаимодействии распределенных приложений предлагается использовать формат JSON.

JSON (*JavaScript Object Notation*) — текстовый формат обмена данными, основанный на языке JavaScript. Был разработан Дугласом Крокфордом в начале 2000-х годов.

Основная идея этого формата — упростить представление данных для несложных объектов и сделать его более наглядным по сравнению с форматами, основанными на языке XML. Его возможности весьма ограничены следующими типами представлений:

а) примитивными типами: ***число***, ***строка***, ***двоичное значение*** и ***null***;

б) двумя структурными типами: ***объекты*** и ***массивы***.

Основные понятия этого формата отображены в таблице 4.3.

Таблица 4.3 — Основные понятия формата представления JSON [17]

Понятие	***Описание***
Число	Десятичное представление, как в языке Java. Восьмеричные и шестнадцатеричные значения не используются.
Строка	Последовательность из нуля или более символов Unicode, ограниченная двойными кавычками и использующая для экранирования символ обратного слеша.
Значение	Представлено в одном из следующих форматов: строка в двойных кавычках, число, двоичное значение, объект или массив.
Массив	Упорядоченный набор значений, заключенный в квадратные скобки, обозначающие начало и конец массива. Значения массива разделены запятыми и могут быть объектами.
Объект	Это — неупорядоченный набор пар «имя/значение», разделенных запятыми и заключенный в фигурные, которые обозначают начало и конец объекта. Пары «имя/значение» соответствуют атрибутам POJO-классов в языке Java.

В Web-технологиях, официальный тип содержимого представления JSON — ***application/json***, а расширение имен файлов — ***.json***.

Для языка Java имеется собственная спецификация ***JSR 353***, которая полностью реализована, начиная с программной платформы Java EE 7, и имеет обозначение API, как ***JSON-P***.

В настоящее время технология JSON является достаточно популярной. Для не на языке Java разработано множество различных инструментов, например, ***JSON-Lib***, ***fastjson***, ***Flexjson***, ***Jettison***, ***Jackson*** и другие. Мы, в данном подразделе, рассмотрим только основы этой технологии, входящей в официаль-

ное программное обеспечение платформы Java EE.

4.2.1 Программное обеспечение технологии JSON

Основные пакеты JSON-P, обслуживающие технологию JSON, представлены в таблице 4.4.

Таблица 4.4 — Основные пакеты API JSON-P [17]

Пакет	***Описание пакета***
javax.json	Предоставляет API для описания структуры данных JSON, таких как: класс JsonArray для массива JSON и класс JsonObject для объекта JSON. Предоставляет точку входа для анализа, построения, чтения и записи объектов и массивов JSON с помощью потоков.
javax.json.spi	Интерфейс провайдера служб (Service Provider Interface), предназначенный для подключения классов, таких как JsonParser и JsonGenerator.
javax.json.stream	Предоставляет потоковый API для анализа и генерации JSON.

Основной API рассматриваемой технологии опирается на базовый класс ***javax.json.Json***. Он содержит методы для создания объектов следующего набора типов:

1. **JsonParser** — класс для обеспечения потокового анализа JSON-документа с использованием сигналов. Создается методом ***createParser(...)*** на потоке чтения JSON-документа.
2. **JsonGenerator** — класс для создания JSON-документа. Создается методом ***createGenerator(...)*** на потоке вывода информации.
3. **JsonWriter** — класс для создания потока вывода JSON-документа.
4. **JsonReader** — класс для создания потока ввода JSON-документа.
5. **JsonObjectBuilder** — класс для создания объекта при построении JSON-документа. Сам создается методом ***createObjectBuilder(...)***.
6. **JsonArrayBuilder** — класс для создания массивов при построении JSON-документа. Сам создается методом ***createArrayBuilder(...)***.

Представление JAVA-объектов в формате JSON требует хорошего знания структуры этих объектов.

Действительно, если технология JAXB использует аннотации, которые устанавливаются перед полями или соответствующими методами класса и, далее, сами преобразования осуществляются автоматически, то технология JSON

требует явного описания программистом всех переменных и имен объектов этих классов. И, чтобы не быть голословными, проведем JSON-представление результата работы программы ***Test4*** проекта ***lab5***, которое сохранено в файле ***lab5.ListLetters.xml*** и показано ранее на рисунке 4.4. Результат такого представления показан на листинге 4.6.

Листинг 4.5 — JSON-представление текста из файла lab5.ListLetters.xml

```
{
	"listLetters":{
		"name":"Список писем для приложения lab5:Test3",
		"list-letters":[
			{
				"list":{
					"id":21,
					"date":"2020-09-06T09:53:25.766+07:00",
					"name":"upk",
					"text":"Сообщение №21 для lab5:Test3"
				}
			},
			{
				"list":{
					"id":22,
					"date":"2020-09-06T09:53:25.766+07:00",
					"name":"asu",
					"text":"Сообщение №22 для lab5:Test3"
				}
			}
		]
	}
}
```

Необходимо иметь достаточно хорошее программистское воображение, чтобы определить в таком достаточно примитивном представлении объекты классов ***ListLetters*** и ***Letter***. Кроме того, возникнут неприятные проблемы с преобразованием переменной ***date***, поскольку программное обеспечение JSON-P не имеет средств для работы с форматом даты.

Если предположить, что в работе используются классы Java с более сложной структурой и большей вложенностью дочерних классов, то привлекательность формата JSON уже не будет столь очевидной.

Для анализа структуры JSON-представлений можно использовать объекты класса ***Parser***, которые обеспечивают потоковую обработку информации. Мы такой подход использовать не будем, а обратим внимание на прямое извлечение информации из JSON-документов. Оно обеспечивается двумя классами:

1. **JsonObject** — класс для извлечения представления объекта из входного потока или из объекта того же класса ***JsonObject***, используя указание имени объекта.
2. **JsonArray** — класс для представления массива объектов, извлекаемое из

объектов класса ***JsonObject***.

Каким образом — это делается? Рассмотрим на конкретных примерах.

4.2.2 Преобразование объекта Java в документ JSON

Продемонстрируем возможности программного обеспечения JSON-P, реализовав представление объекта класса ***ListLetters*** в формат JSON, как это представлено на листинге 4.5, внеся следующие корректировки и дополнения:

а) дату класса ***Letter*** будем отображать числом типа ***Long***;

б) пример преобразования реализуем в новом проекте ***lab6*** типа ***Dynamic Web Project***, чтобы подключить программное обеспечение сервера приложений TomEE, содержащее инструменты пакета JSON-P;

в) приложение назовем ***Test1***;

г) результат преобразования сохраним в файле ***$HOME/lab6.Test1.json***;

д) для преобразования воспользуемся методами класса ***JsonObjectBuilder***.

Результат реализации приложения ***Test1*** представлен на листинге 4.6.

Листинг 4.6 — Исходный текст приложения Test1.java проекта lab6

```
package rsos.lab6;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;

import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;

public class Test1
{
      public static void main(String[] args) throws IOException
      {
            // Создание основного инструментального объекта
            JsonObjectBuilder job =
                        Json.createObjectBuilder();

            // Преобразование текущей даты в тип Long
            Long tt =
                        new Date().getTime();
            // Представление типа Long в виде строки
            String ss =
                         tt.toString();

            // Формирование представления согласно листинга 4.5
            JsonObject obj =
                  job.add("listLetters", Json.createObjectBuilder()
                     .add("name", "Список писем для приложения lab5:Test3")
                     .add("list-letters", Json.createArrayBuilder()
                     .add(Json.createObjectBuilder()
```

```
                        .add("list", Json.createObjectBuilder()
                        .add("id", 21)
                        .add("date", ss)
                        .add("name", "upk")
                        .add("text", "Сообщение №21 для lab5:Test3")))
                        .add(Json.createObjectBuilder()
                        .add("list", Json.createObjectBuilder()
                        .add("id", 22)
                        .add("date", ss)
                        .add("name", "asu")
                        .add("text", "Сообщение №22 для lab5:Test3")))))
                        .build();

                // Открытие потока вывода в файл
                FileWriter writer =
                      new FileWriter(new File(System.getenv("HOME" )
                                        + "/lab6.Test1.json"));

                // Запись результата в файл
                writer.write(obj.toString());
                writer.close();
        }
}
```

Запустив приложение ***Test1*** на выполнение, мы получим заявленный файл ***$HOME/lab6.Test1.json*** с содержимым, показанным на рисунке 4.5.

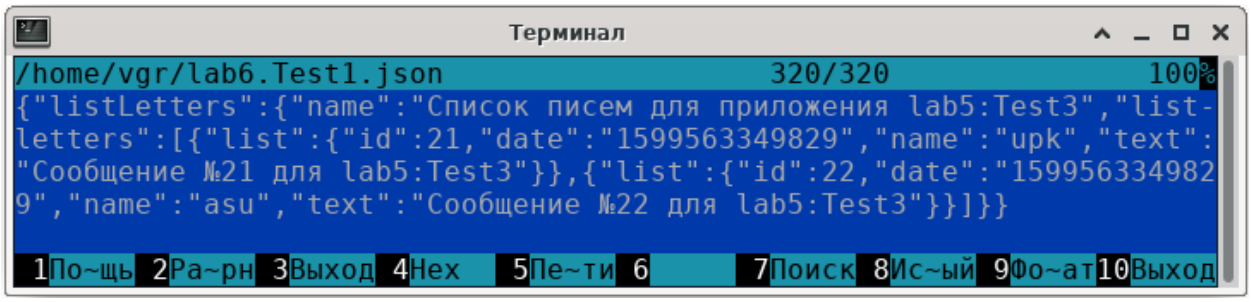

Рисунок 4.5 — Результат работы приложения Test1 проекта lab6

Хорошо видно, что результат работы программы ***Test1*** представляет собой неформатированную текстовую строку, но даже в таком виде ее можно вполне корректно сопоставить с текстом исходного листинг 4.5. При более сложном объекте представления, например, при десяти записях объектов класса ***Letter***, сопоставление таких представлений вызывало бы серьезные проблемы.

Потребитель представления в формате JSON должен адекватно выполнять обратное преобразование.

Теперь рассмотрим обратное преобразование текущего содержимого файла ***$HOME/lab6.Test1.json*** в объект класса ***ListLetters***. Для этого:

1) создадим в проекте ***lab6*** классы ***Letter*** и ***ListLetters***, позаимствовав их сожержимое из проекта ***lab5***; аннотации из классов можно убрать;
2) создадим приложение ***Test2***, приведенное на листинге 4.7, которое извлекает информацию из файла результата приложения ***Test1*** и создает объект класса ***ListLetters***.

Листинг 4.7 — Исходный текст приложения Test2.java проекта lab6

```
package rsos.lab6;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.json.Json;
import javax.json.JsonArray;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonValue;

public class Test2
{
	public static void main(String[] args) throws FileNotFoundException
	{
		// Открываем файл для чтения
		JsonReader reader =
			Json.createReader(new FileReader(System.getenv("HOME" )
					+ "/lab6.Test1.json"));

		// Читаем содержимое файла в объект класса JsonObject
		JsonObject obj =
				reader.readObject();
		reader.close();

		// Выделяем корневой класс и приступаем к созданию объекта
		obj =
			obj.getJsonObject("listLetters");

		// Создаем целевой объект
		ListLetters listlets =
				new ListLetters(obj.getString("name"));

		// Выделяем объекты массива
		JsonArray array =
				obj.getJsonArray("list-letters");

		// Создаем пустой список
		List<Letter> list =
				new ArrayList<>();

		// Рабочие переменные
		Letter let;
		Long ll;
```

```
            // В цикле создаем объекты класса Letter
            for(JsonValue value : array)
            {
                  // Преобразуем JsonValue в JsonObject
                  obj =
                        value.asJsonObject();

                  // Выбираем объект Java-класса Letters
                  obj =
                        obj.getJsonObject("list");
                  ll =
                        Long.valueOf(obj.getString("date"));

                  // Создаем объект Java-класса Letters
                  let =
                        new Letter(new Date(ll.longValue()),
                                    obj.getString("name"),
                                    obj.getString("text"));
                  let.setId(obj.getInt("id"));

                  // Добавляем объект в список
                  list.add(let);
            }

            // Полностью создаем объект Java-класса ListLetters
            listlets.setList(list);

            // Печатаем объект на консоль
            System.out.println(listlets.toString());
      }
}
```

На рисунке 4.6 представлен результат запуска приложения ***Test2***.

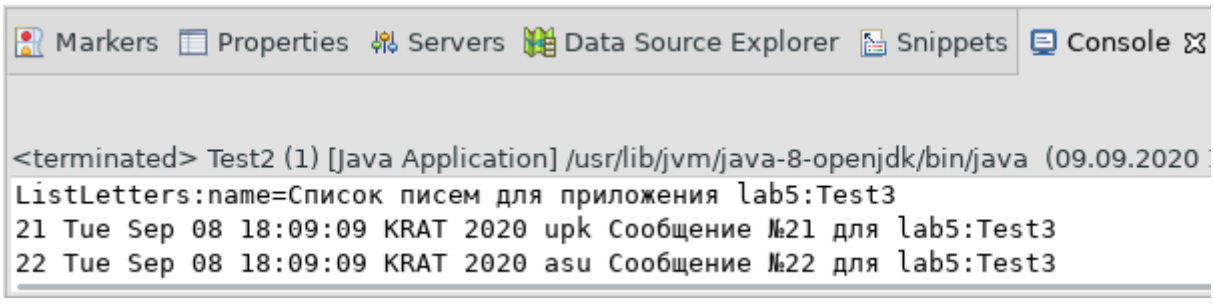

Рисунок 4.6 — Результат работы приложения Test2 проекта lab6

4.2.3 Пример представления JSON на уровне классов

В простых классах языка Java, гораздо проще использовать собственные методы преобразования объектов в отображение JSON.

Примеры приложений ***Test1*** и ***Test2***, изложенные в предыдущем пункте,

показывают, что технология применения JSON не столь приятна в упот-реблении, в противовес тому, как она рекламируется. В данном пункте, мы рассмотрим подход, предполагающий добавление к каждому классу метода, который бы делал форматированное отображение объектов класса в формат JSON.

Идея данного подхода полностью аналогична идее переопределения метода ***toString()***, что используется для текстового представления содержимого переменных класса. Отличие состоит в двух пунктах:

- метод будет называться ***String toJson(String blank)***;
- строковый аргумент функции — ***blank*** предназначен для задания начального отступа текста представления, использованый для целей форматирования.

Данную идею применим к классам ***Letter*** и ***ListLetter***, а затем продемонстрирует приложениями ***Test3*** и ***Test4***, которые по своей сути аналогичны приложениям ***Test1*** и ***Test2***.

Реализацию идеи начнем с класса ***Letter***, в исходный текст которого добавим метод, представленный на листинге 4.8.

Листинг 4.8 — Метод toJson(…) класса Letter.java проекта lab6

```
// Преобразование объекта в формат JSON
public String toJson(String blanks)
{
        // Преобразование даты в тип Long
        Long tt =
                    new Date().getTime();

        // Форматированное представление объекта
        return blanks + "{\n"
         + blanks + "\t\"Letter\":{\n"
         + blanks + "\t\t\"id\":" + Integer.toString(id) + ",\n"
         + blanks + "\t\t\"date\":\"" + tt.toString()  + "\",\n"
         + blanks + "\t\t\"name\":\"" + name + "\",\n"
         + blanks + "\t\t\"text\":\"" + text + "\"\n"
         + blanks + "\t}\n"
         + blanks + "}";
}
```

Хорошо видно, что для небольшого класса написать и отладить JSON-представление — гораздо проще, чем — для большого.

Теперь реализуем метод ***toJson(...)*** для класса ***ListLetters***, что показано на листинге 4.9.

Листинг 4.9 — Метод toJson(…) класса ListLetters.java проекта lab6

```
// Преобразование объекта класса ListLetters в формат JSON.
// Также используется аналогичный метод для класса Letter.
public String toJson(String blanks)
```

```
    {
            // Часть преобразования до списка объектов класса Letter.
            String ss =
                     blanks + "{\n"
                   + blanks + "\t\"ListLetters\":{\n"
                   + blanks + "\t\t\"name\":\"" + name + "\",\n"
                   + blanks + "\t\t\"list\":[\n";

            // Определяем количество записей в списке.
            int n =
                  list.size();

            // В цикле вставляем отображения класса Letter.
            for(Letter letter : list)
            {
                  n--;
                  ss +=
                        (blanks + letter.toJson("\t\t\t"));
                  if(n > 0) ss += ",\n";
                  else      ss += "\n";
            }

            // Завершаем формирование и вывод отображения.
            return
                  ss + blanks + "\t\t]\n"
                     + blanks + "\t}\n"
                     + blanks + "}\n";
    }
```

В приведенном листинге используется уже реализованный ранее метод ***toJson(...)*** для класса ***Letter***. Обратите внимание, что сложность реализации метода не зависит от количества вставленных объектов класса ***Letter***.

Теперь представим на листинге 4.10 реализацию демонстрационного приложения ***Test3***, формирующего объект класса ***ListLetters*** со списком из двух объектов класса ***Letter***, записывает полученное отображение на консоль системы разработки и в файл ***$HOME/lab6.Test3.json***.

Листинг 4.10 — Исходный текст класса Test3.java проекта lab6

```
package rsos.lab6;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Test3
{
      public static void main(String[] args) throws IOException
      {
            // Создаем список из двух объектов класса Letter
            List<Letter> lets = new ArrayList<>();
```

```
            // Первый объект класса Letter
            Letter letter =
                  new Letter(new Date(), "upk",
                              "Сообщение №21 для lab6:Test3");
            letter.setId(21);

            // Добавляем объект в список
            lets.add(letter);

            // Второй объект класса Letter
            letter =
                  new Letter(new Date(), "asu",
                              "Сообщение №22 для lab6:Test3");
            letter.setId(22);

            // Добавляем объект в список
            lets.add(letter);

            // Создаем объект класса ListLetters
            ListLetters listlets =
                  new ListLetters("Список писем для приложения lab6:Test3");
            listlets.setList(lets);

            // Выводим на консоль
            System.out.println(listlets.toString() + "\n");
            System.out.println(listlets.toJson("") + "\n");

            // Выводим в файл
            FileWriter writer =
                  new FileWriter(new File(System.getenv("HOME" )
                              + "/lab6.Test3.json"));
            writer.write(listlets.toJson(""));
            writer.close();
      }
}
```

Как видим, основной текст листинга 4.10 состоит из команд формирования самого объекта ***ListLetters***, а преобразование в формат JSON осущест-вляется вызовом только одного метода.

Результат работы приложения ***Test3*** показан на рисунке 4.7.

Как видим, представление объекта класса ***ListLetter*** в формат JSON — хорошо форматировано и имеет адекватное представление, соотвествующее реальным именам используемых классов и их переменных. Это обеспечивает гораздо большую надежность реализации таких представлений, чем прямое программирование JSON-представлений объектов сложных классов с помощью средств класса ***JsonObjectBuilder***.

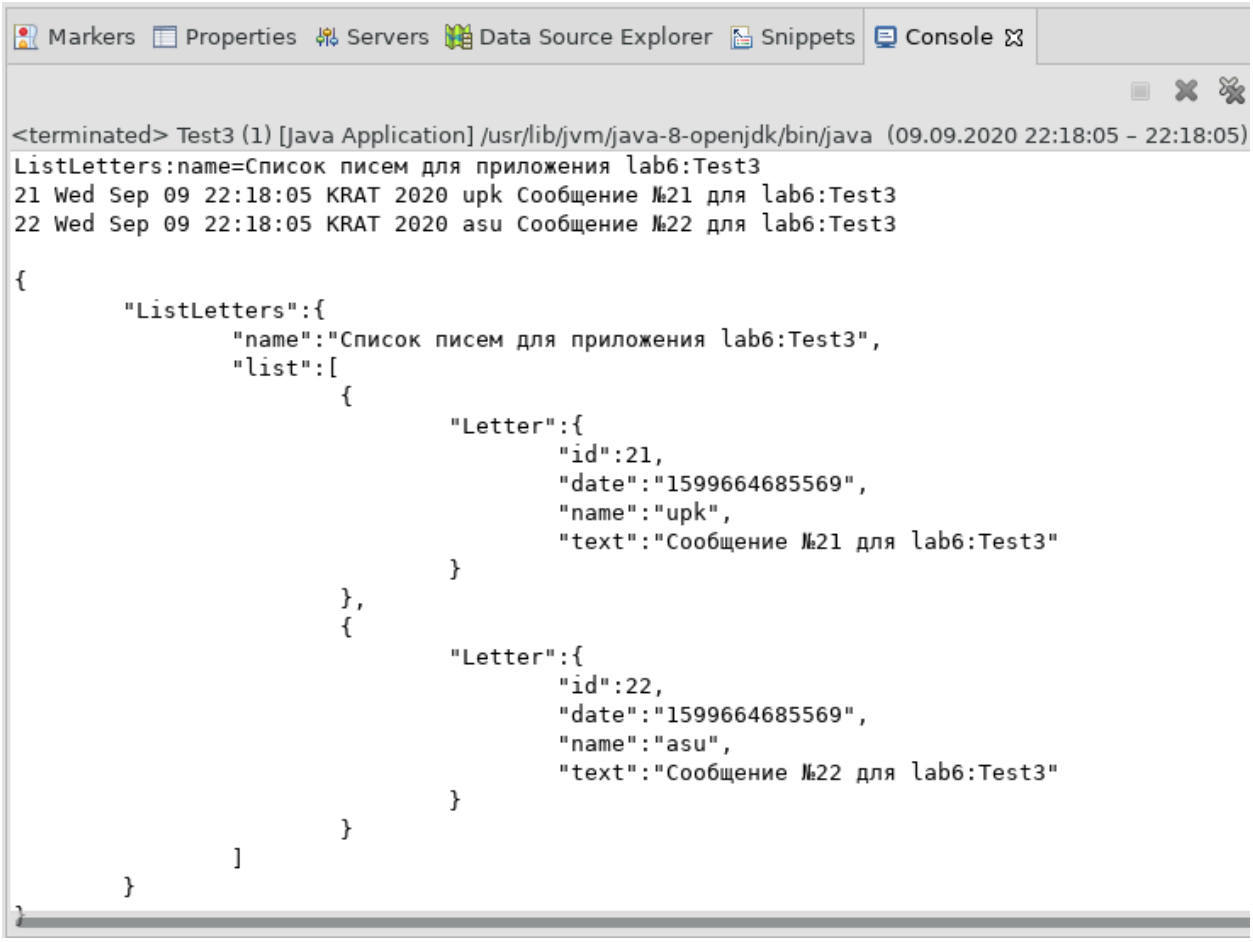

Рисунок 4.7 — Результат работы приложения Test3 проекта lab6

Теперь убедимся, что данное форматированное представление обеспечивает обратное преобразование в соотвествующий объект класса ***ListLetters***. Для этого, реализуем приложение ***Test4***, иходный текст которого показан на листинге 4.11 и полностью аналогичен исходному тексту приложения ***Test2***, с точностью до конкретных используемых имен представления, записанного в файл ***$HOME/lab6.Test3.json***.

Листинг 4.10 — Исходный текст класса Test4.java проекта lab6

```
package rsos.lab6;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.json.Json;
import javax.json.JsonArray;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonValue;
```

```
public class Test4
{
	public static void main(String[] args) throws FileNotFoundException
	{
		// Открываем файл для чтения
		JsonReader reader =
			Json.createReader(new FileReader(System.getenv("HOME" )
					+ "/lab6.Test3.json"));

		// Читаем содержимое файла
		JsonObject obj =
				reader.readObject();
		reader.close();

		// Выделяем корневой класс и приступаем к созданию объекта
		obj =
			obj.getJsonObject("ListLetters");

		// Создаем объект
		ListLetters listlets =
				new ListLetters(obj.getString("name"));

		// Выделяем объекты массива
		JsonArray array =
					obj.getJsonArray("list");

		// Создаем пустой список
		List<Letter> list =
					new ArrayList<>();

		// Рабочие переменные
		Letter let;
		Long ll;

		// В цикле создаем объекты класса Letter
		for(JsonValue value : array)
		{
			obj =
				value.asJsonObject();
			obj =
				obj.getJsonObject("Letter");
			ll =
				Long.valueOf(obj.getString("date"));
			let =
				new Letter(new Date(ll.longValue()),
					obj.getString("name"),
				obj.getString("text"));
			let.setId(obj.getInt("id"));

			// Добавляем объект в список
			list.add(let);
		}

		// Полностью создаем объект
		listlets.setList(list);
```

```
            // Печатаем объект на консоль
            System.out.println(listlets.toString());
        }
}
```

Результат работы приложения ***Test4*** показан на рисунке 4.8.

Сравнивая изображения рисунков 4.7 и 4.8, мы видим полную идентичность исходного и восстановленного из файла представлений объекта класса ***ListLetters***.

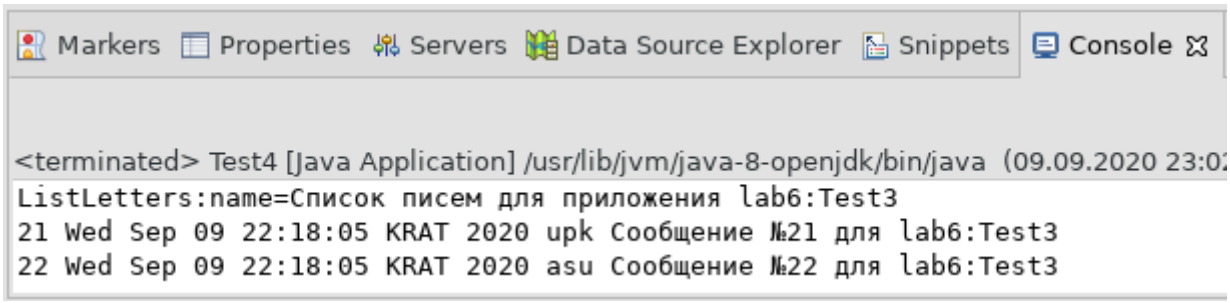

Рисунок 4.8 — Результат работы приложения Test4 проекта lab6

4.2.4 Выводы по результатам изучения главы 4

В данной главе были кратко описаны две технологии: JAXB и JSON. Обе они связаны с представлениями данных объектов языка Java в соотвествующие текстовые форматы и обратным преобразованием из текстовых строк в объекты.

С точки зрения технологии применения этих форматов следует отметить, что JAXB предоставляет высокоуровневые средства программной платформы Java EE, ориентированные на использование аннотаций. В этом отношении, JAXB является универсальным инструментом преобразований, обеспечивающим преобразование произвольных сериализуемых POJO-классов.

С другой стороны, технология JXON выступает как очень популярная альтернатива JAXB. Она ограничена поддержкой только нескольких простых типов данных Java и требует от IT-разработчиков применения низкоуровневых средств программирования.

Обе технологии прямо ассоциируются с рассмотренной в предыдущей главе технологией JPA, поскольку связаны с преобразованием и хранением данных объектов языка Java.

В плане тематики учебного процесса изучаемой дисциплины, данная глава завершает предварительное изучение общих технологий программной платформы языка Java, необходимых для последующего непосредственного изучения технологий сервис-ориентированных систем.

Вопросы для самопроверки

1. Перечислите основные понятия, которые используются для формата представления на языке XML?
2. Что общего между технологиями JPA и JTA, с одной стороны, и технологиями XML и JSON, с другой стороны?
3. Что такое — маршалинг и демаршалинг JAVA-объектов?
4. Для чего используется класс JAXBContext?
5. Для чего используется аннотация @XmlRootElement?
6. Для чего используется аннотация @XmlTransient?
7. Где устанавливаются аннотации @XmlElement и @XmlAttribute, если переменные аннотируемого класса имеют спецификацию доступа private?
8. Какую аннотацию необходимо применить, чтобы указать точное следование переменных JAVA-класса в отображении XML?
9. Зачем некоторым классам языка Java необходимо реализовывать интерфейс Serializable?
10. Для чего в аннотациях технологии JAXB используется параметр name?
11. Назовите основные понятия формата представления JSON?
12. Как обозначается объект в представлении JSON?
13. Как обозначается массив в представлении JSON?
14. Какие четыре примитивных типа поддерживает технология JSON?
15. Для каких целей нужно использовать объекты классов JsonObject и JsonArray?

5 Тема 5. Web-службы SOAP

Классика сервис-ориентированных систем — Web-службы SOAP.

В пункте 1.2.2 первой главы был рассмотрен вопрос: «*Развитие концепции SOA*». Было отмечено, что к началу 2000-х годов были сформированы Web-сервисы первого поколения, включающие: язык описания интерфейсов WSDL, протокол взаимодействия клиента сервиса и поставщика сервиса SOAP, а также инструмент для размещения описаний описаний WSDL — UDDI.

Принципиальной особенностью этой технологии было тотальное использование языка XML, на основе которого и был создан язык WSDL. Когда в 1998 году Дейв Винер опубликовал протокол XML-RPC, модифицированный через несколько лет в протокол SOAP, Web-службы первого поколения были приравнены к сервис-ориентированным технологиям, на основе которых стали разрабатываться сервис-ориентированные системы с общей архитектурой взаимодействия, показанной на рисунке 5.1.

Рисунок 5.1 — Общая архитектура взаимодействия базовых составляющих сервис-ориентированных систем [17]

Заметим, что термины ***сервис*** и ***служба***, в данном контексте, имеют одно и то же семантическое значение и соответствуют одному англоязычному термину ***service***, поэтому мы их также различать не будем. Далее, учитывая достаточную учебную подготовку, выполненную нами ранее, учебный материал данной главы сосредоточим на изучении следующих трех вопросов: опишем основные

составляющие Web-служб, а затем — рассмотрим создание поставщиков и потребителей этого классического сервиса.

5.1 Основные составляющие Web-служб SOAP

В предыдущей главе мы убедились с какой «*легкостью*» технология JAXB обеспечивает преобразование объектов языка Java в XML-представление и обратно. Но эта легкость является обманчивой, поскольку мы использовали технологический стиль аннотаций, а также не рассматривали XML-схемы — ***XSD***, предназначенные для контроля типов данных и синтаксических конструкций реализованных описаний на языке XML.

Основная учебная цель данного подраздела — краткая характеристика языка ***WSDL***, протокола ***SOAP*** и необязательного реестра служб (***UDDI***), технологически основанных на языке ***XML***.

5.1.1 Протоколы и языки Web-служб

Появившиеся в 1990 году Web-технологии предоставили обществу три технологические новинки: универсальную адресацию ***URI*** (*URL*), язык разметки гипертекста ***HTML*** и приложение-***браузер***, обеспечивающее простейшее графическое представление текста и рисунков в формте ***GIF***.

В пункте 1.2.1 первой главы, мы уже рассматривали этот вопрос. Здесь лишь отметим тот факт, что две другие составляющие Web-технологий — Web-сервер и протокол HTTP большими новинками не являлись рассматривались как самособой разумеющееся дополнение к узананным выше трем.

Ситуация существенно стала меняться с развитием сетевой инфраструктуры и широким распространением приложений-браузеров. Тогда нео-жидано выяснилось, что новая технология просто идеально подходит для нового технологического направления — сервис-ориентированных систем. Действительно:

а) появилась сетевая инфраструктура, которая прямо предназначена для решения слабосвязанных распределенных задач;

б) появились серверные технологии, такие как технология динамического формирования страниц ***PHP*** и технология ***JavaServer Pages*** (*JSP*), обеспечивающие формирование трехзвенных распределенных архитектур приложений.

Общая тенденция развития современных Web-технологий — формирование Web-служб.

Все попытки полностью заменить язык разметки гипертекста HTML на более строгий язык XHTML, основанный на языке XML, полностью потерпели неудачу. Настоящим и возможно окончательным вариантом назван язык HTML версии 5. При внимательном анализе ситуации мы приходим к пониманию, что строгие синтаксические конструкции, обеспечиваемые языком XML, нужны приложениям разработанным, например, на языке Java, а не браузерам, разработчики которых научились динамически исправлять синтаксические ошибки языка HTML.

Язык XML, при всей его теоретической строгости, является для многих приложений избыточным, а главное — плохо читаемым, поскольку полностью в явном виде отображает все присущие описаниям связи. Благодаря этим его «*недостаткам*» имеется общая тенденция отказа на платформе Java EE от дескрипторов развертывания и замена их на статические описания с помощью аннотаций. Указанная тенденция наглядно была продемонстрирована учебными примерами предыдущих глав данного пособия.

Что касается протокола HTTP, то он является не единственным средством, поверх которого может работать протокол ***SOAP***.

Таким образом, мы видим, что между Web-технологиями и Web-службами, в концепции сервис-ориентированных систем, имеется достаточно гибкая связь, которая не мешает достаточно независимо развиваться каждой из них.

Учитывая сказанное выше, раскроем более подробно следующие три технологических аспекта Web-служб:

1. **WSDL** (*Web Services Description Language*) — язык описания Web-служб, определяющий протокол, интерфейс, типы сообщений и взаимодействие между потребителем и поставщиком сервиса.
2. **SOAP** (*Simple Object Access Protocol*) — простой протокол доступа к объекту, обеспечивающий кодирование сообщений на основе технологий XML и определяющий «*конверт*» для общения Web-служб.
3. **UDDI** (*Universal Description Discovery and Integration*) — необязательный реестр служб и механизмов обнаружения поставщиков сервисов, похожий на телефонный справочник, который может быть использован для хранения и категоризации SOAP-интерфейсов Web-служб (***WSDL***).

5.1.2 Краткое описание языка WSDL

WSDL — это язык определения интерфейса, наподобие IDL технологии CORBA, который определяет взаимодействие между потребителями сервиса и веб-службами SOAP. Он описывает типы передаваемых сообщений, порты со-

единений, протоколы коммуникаций, поддерживаемые операции и то, что потребитель сервиса должен получить взамен своего запроса.

В терминологии языка Java, WSDL определяет контракт, которому будет соответствовать Web-служба. Его также можно рассматривать как представление интерфейса языка Java, написанного на языке XML. Нетрудно догадаться, что такое описание должно быть большим и сложным.

В целом, взаимодействие между потребителем и поставщиком Web-сервиса, учитывая, что UDDI не является обязательным участником взаимодействия, можно представить рисунком 5.2.

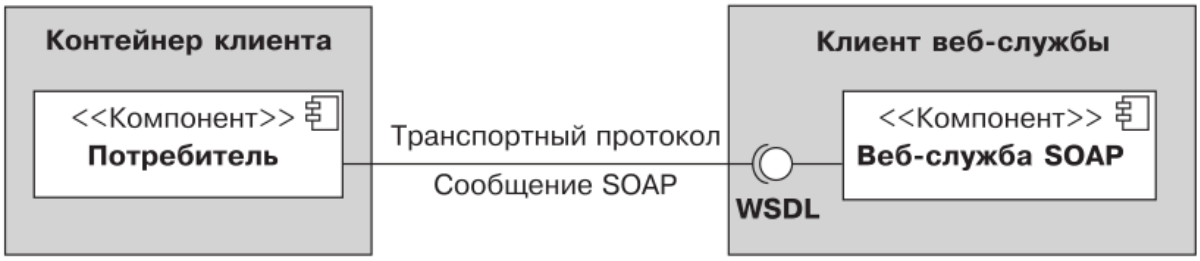

Рисунок 5.2 — Интерфейс WSDL между потребителем и Web-службой [17]

Главное отличие технологии использования языка WSDL от налогичного использования языка IDL технологии CORBA состоит в том, что сервер Web-службы публикует свой интерфейс и потребитель сервиса всегда его может прочитать.

На листинге 5.1 представлен шаблон общей структуры WSDL-документа.

Листинг 5.1 — Шаблон общей структуры WSDL-документа

```
<definitions>
   <types>
      Определение типов........
   </types>

   <message>
      Определение сообщений....
   </message>

   <portType>
      <operation>
         Определение операций.......
      </operation>
   </portType>

   <binding>
```

```
      Определение связывания....
   </binding>

   <service>
      Определение сервиса....
   </service>
</definitions>
```

Кратко перечислим назначение основных элементов и атрибутов языка WSDL, заданных в виде следующих тегов:

<definitions> — корневой элемент WSDL, определяющий глобальные описания пространств имен, которые видны на протяжении всего документа.

<types> — пределяет типы данных, которые будут использованы в сообщениях.

<message> — определяет формат данных, которые передаются между потребителем Web-службы и самой Web-службой. Они также разделены на запросы и ответы.

<portType> — определяет операции Web-служб (методы), причем каждая операция ссылается на входное и выходное сообщение.

<binding> — описывает конкретный протокол SOAP и форматы данных для операций и сообщений, определенных для конкретного типа порта.

<service> — содержит коллекцию элементов ***<port>***, где каждый порт связан с конечной точкой (сетевым адресом или URL).

<port> — указывает адрес для связывания, таким образом, определяя конечную точку коммуникации.

В целом, WSDL описывает абстрактный интерфейс веб-службы.

5.1.3 Краткое описание протокола SOAP

SOAP предоставляет конкретную реализацию взаимодействия потребителей и поставщиков сервиса, определяя XML-сообщения, которыми обмениваются потребитель и поставщик.

SOAP предназначен для обеспечения независимого, абстрактного протокола связи, который обладает возможностью подключения распределенных Web-служб.

На листинге 5.2 представлен шаблон общей структуры SOAP-сообщения или структура конверта SOAP.

Листинг 5.2 — Шаблон общей структуры SOAP-сообщения

```
<?xml version = "1.0" encoding="utf-8"?>
<soap:Envelope
           xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
      ...">

   <soap:Header>
      ...
      ...
   </soap:Header>
   <soap:Body>
      ...
      ...
      <soap:Fault>
         ...
         ...
      </soap:Fault>
      ...
   </soap:Body>
</soap:Envelope>
```

В структуре сообщений SOAP четко выделяются четыре элемента:

Envelope — обязательный корневой элемент, определяющий сообщение и пространство имен, использованное в документе.

Header — содержит любые необязательные атрибуты сообщения или характерную для приложения инфраструктуру, например, такую как информация о безопасности или о сетевой маршрутизации.

Body — основной обязательный элемент, содержащий сообщение, которым обмениваются приложения.

Fault — необязательный элемент, используемый для предоставления информации об ошибках, произошедших при обработке сообщений.

5.1.4 Необязательный реестр Web-служб — UDDI

Необязательный реестр Web-служб — UDDI призван завершить инфраструктуру технологии Web-сервисов, построенных на протоколе SOAP.

Основная задача UDDI — регистрировать и публиковать для общего использования интерфейсы поставщиков Web-сервисов. После того как потребители сервисов выбрали нужного поставщика, они напрямую взаимодействуют друг с другом по протоколу SOAP.

Нетрудно заметить, что эта идея тесно коррелирует с идеей брокеров, входящих в инфраструктуру объектного подхода CORBA, используемого в распре-

деленных системах. Тем не менее, несмотря на значительные усилия основных «*идеологов*» сервисных Web-технологий, к которым относятся корпорации IBM, Microsoft и SAP, использование UDDI не получило широкого распространения. В январе 2006 года ряд компаний объявили о закрытии своих общедоступных реестров UDDI. А в конце 2007 года, рабочая группа по UDDI в OASIS объявила о закрытии своего технического комитета.

Таким образом, технология UDDI была признана устаревшей, выведенной из состава программной платформы Java EE и мы не будем ее более рассматривать в данной дисциплине.

5.1.5 Программные пакеты Java EE, обслуживающие SOAP

Глобальный консорциум ***OASIS*** (*Organization for the Advancement of Structured Information Standards*) хранит множество официальных спецификаций, относящихся к технологии Web-сервисов (*Web-служб*). Все они опубликованы под общим префиксом ***WS-****.

В языке Java разработка и утверждение всех спецификаций проходит под контролем процесса ***JCP*** (*Java Community Process*) — сформированного в 1998 году. JCP выпускает спецификации JSR под разными номерами, включая те, которые относятся к Web-сервисам.

В настоящее время, основу технологии Web-сервисов представляют спецификации:

а) ***JAX-WS 2.2a*** — спецификация JSR 224;

б) ***Web Services 1.3*** — спецификация JSR 109;

в) ***Web Services Metadata 2.3*** — спецификация JSR 181;

г) ***JAXB 2.2*** — спецификация JSR 222.

Технологию JAXB мы рассмотрели в предыдущей главе, а технология Web Services 1.3 определяет модель программирования и поведение программного обеспечения во время выполнения Web-служб в контейнере Java EE и упаковку артефактов для обеспечения переносимости Web-служб на разные реализации серверов.

Технология JAX-WS 2.2a определяет набор API и основные аннотации, позволяющие создавать и использовать Web-службы в Java. Ее пакеты представлены в таблице 5.1. В них определяются API для различных обработчиков сообщений, поэтому, ни потребитель сервиса, ни Web-служба не должны сами генерировать или анализировать SOAP-сообщения. Этой низкоуровневой обработкой занимается программное обеспечение JAX-WS. В частности, JAX-WS интенсивно использует технологию JAXB, «*скрывая*» от участников взаимодействия все детали достаточно сложных преобразований.

Таблица 5.1 — Основные пакеты технологии JAX-WS 2.2a [17]

Пакет	***Описание пакета***
javax.xml.ws	В этом пакете содержатся основные API JAX-WS.
javax.xml.ws.http	Определяет API, характерные для связывания XML/HTTP.
javax.xml.ws.soap	Определяет API, характерные для связывания SOAP 1.1/HTTP или SOAP 1.2/HTTP.
javax.xml.ws.handler	В этом пакете определяются API для обработчиков сообщений.

Технолгия WS-Metadata 2.3 предоставляет аннотации, помогающие определять и развертывать Web-службы. Основная цель WS-Metadata — упростить развитие Web-служб, обеспечивая правильное и надежное отображение используемых объектов между описаниями WSDL и интерфейсами Java. Основные пакеты этой технологии представлены в таблице 5.2.

Таблица 5.2 — Основные пакеты технологии WS-Metadata 2.3 [17]

Пакет	***Описание пакета***
javax.jws	Содержит аннотации для перехода от Java к WSDL и обратно.
javax.jws.soap	API, предназначенные для преобразования Web-служб к протоколу передачи сообщений SOAP.

Часто, все перечисленные выше технологи, неофициально обозначают как JWS (*Java Web Services*) или просто — Web-службы Java. Некоторые из этих технологий были перенесены на уровень Standard Edition и влючены в Runtime-систему языка Java. Другие, такие как JAX-RPC (JSR 101), на основе которых создавались технологии протокола XML-RPC, были удалены из платформы Java EE по причине их излишней громоздкости и сложности. Тоже самое произошло и с технологией JAXR (*Java API for XML Registers*), обслуживавшей доступ к службе реестров UDDI и удалена по причине их бесперспективности.

Особо следует отметить набор утилит, присутствующий в Runtime-системе языка Java. Так, в каталоге ***/usr/lib/jvm/default-runtime/bin*** учебного дистрибутива ОС УПК АСУ, вы найдете утилиты:

а) ***wsgen*** — читает класс реализации Web-сервиса (*SEI, Service Endpoint Implementation*) и создает все необходимые артефакты для развертывания Web-сервиса и его вызова;

б) ***wsinport*** — генерирует на основе описания WSDL необходимые артефакты технологии JAX-WS, которые затем могут быть запакованы в WAR-архив и размещены на сервере приложений.

Дальнейшее изучение технологии Web-сервисов необходимо привязывать к конкретным примерам.

5.2 Создание Web-служб SOAP

Web-службы SOAP создаются и обеспечиваются провайдерами сервисов, в общем случае, независимо от потребителей сервисов.

В качестве демонстрационного примера выберем EJB-приложение, которое было реализовано в главе 3 данного пособия, когда изучаласть технология JPA. Напомню, что данное приложение демонстрировало работу с базой данных ***lab4db***, где информация храниться, а также обрабатывается в таблице ***t_letter*** (см. сценарий создания базы данных ***lab4db*** на рисунке 3.5, стр. 149) с помощью двух классов:

а) класс ***Letter*** — класс сущности, выполняющий ORM-отображение переменных объектов класса ***Letter*** в поля таблицы ***t_letter***;

б) класс ***Lets2*** — полная реализация EJB-приложения, обеспечивающего выполнение операций чтения списка всех хранящихся в базе данных объектов класса ***Letter***, а также выполнение стандартных операций изменения данных таблицы ***t_lerrer***: добавление, удаление, модификацию и извлечение отдельных объектов.

Чтобы не создавать путаницы с предыдущими и последующими примерами, демонстрация создания Web-службы SOAP проводится в отдельном проекте с именем ***lab7*** и описание процесса реализации этой Web-службы разделено на четыре части по пунктам:

1) пункт 5.2.1 — подготовка проекта ***lab7***, куда переносятся уже готовые части программного обеспечения из проекта ***lab4***;
2) пункт 5.2.2 — описываются общие аннотации, используемые поставщиками сервиса и одновременно реализуется сам пример Web-службы;
3) пункт 5.2.3 — приводится информация по обработке исключений, возникающих в программном обеспечении Web-службы;
4) пункт 5.2.4 — приводится информация по обработке контекста, реализованной Web-службы.

5.2.1 Подготовка проекта lab7

Подготовку проекта проведем в виде прямой последовательности шагов, претендующих на начальный шаблон реализации всех масштабных проектов.

Шаг 1. Создание проекта типа ***Dynamic Web Project*** с именем ***lab7*** и подключение к нему сервера приложений Apache TomEE.

Создание проекта проводится стандартными средствами инструментальной среды Eclipse EE, с указанием того, что система должна сформировать

дескриптор развертывания ***web.xml***. Новый проект и новое имя проекта нужно для того чтобы разделить уже созданный софт и максимально исключить все зависимости и ньюансы уже созданных приложений.

Шаг 2. В созаданном проекте открываем класс с именем ***Letter*** и именем пакета ***rsos.lab7***.

Напомню, что класс ***Letter*** представляет полностью функционирующую сущность для работы с таблицей ***t_letter*** базы данных ***lab4db***, поэтому будем использовать ее как основу для создания будущего Web-сервиса, а поскольку этот класс уже реализован в проекте ***lab4***, то копируем из него в новый проект ***lab7*** следующие файлы:

1) ***Letter.java*** — правим в нем имя пакета на ***rsos.lab7***;
2) ***derbyclient.jar*** — драйвер для работы с СУБД Apache Derby;
3) ***resources.xml*** — десриптор ресурсов для соединения с СУБД Apache Derby;
4) ***pesistence.xml*** — дескриптор описания юнитов технологи JPA для работы с СУБД Apache Derby.

В файле ***pesistence.xml*** правим ссылки на класс ***Letter***, как это показано на листинге 5.3.

Листинг 5.3 — Исходный текст файла persistence.xml проекта lab7

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

     <persistence-unit name="lab4-unit1" transaction-type="RESOURCE_LOCAL">

          <non-jta-data-source>lab4DerbyUnmanaged</non-jta-data-source>

          <class>rsos.lab7.Letter</class>

    </persistence-unit>

     <persistence-unit name="lab4-unit2" transaction-type="JTA">

                    <jta-data-source>lab4Derby</jta-data-source>
          <non-jta-data-source>lab4DerbyUnmanaged</non-jta-data-source>

          <class>rsos.lab7.Letter</class>

    </persistence-unit>
</persistence>
```

После проведенных изменений, сущность в виде класса ***Letter*** полностью готова для использования ее любым EJB-компонентом программной платформы Java EE.

Шаг 3. Подготовка EJB-компонента перед реализацией Web-сервиса.

За основу функционала будущего Web-сервиса возьмем EJB-приложение ***Lets2***, реализованное в проекте ***lab4*** на основе описаний двух интерфейсов: ***LocalLetter*** и ***RemoteLetter***.

Чтобы не порождать конфликтов между активными проектами, которые могут возникнуть при работе с классами и интерфейсами, имеющими одинаковые имена, создадим в проекте ***lab7***:

1) интерфейс ***RemoteLets***, описывающий все методы EJB-приложения и показанный на листинге 5.4;
2) класс ***Lets7***, реализующий методы интерфейса ***RemoteLets***, как EJB-компонент без сохранения состояния, и показанный на листинге 5.5.

Листинг 5.4 — Исходный текст интерфейса RemoteLets.java проекта lab7

```
package rsos.lab7;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface RemoteLets
{
	// Получить список писем
	List<Letter> getList();

	// Получить письмо по идентификатору
	Letter getLetter(int id);

	// Добавить письмо
	void addLetter(Letter letter);

	// Удалить письмо по идентификатору
	void deleteLetter(int id);

	// Модифицировать письмо
	void modLetter(Letter letter);
}
```

Листинг 5.5 — Исходный текст EJB-компоненты Lets7.java проекта lab7

```
package rsos.lab7;
import java.util.List;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
```

```java
@Stateless
@LocalBean
public class Lets7 implements RemoteLets
{
	@PersistenceContext(name = "lab4-unit2")
	private EntityManager em;

	// Получение списка записей таблицы t_letter
	public List<Letter> getList()
	{
		//System.out.println("Lets7:getList()...");
		// Этап 1. Создаем объект builder
		CriteriaBuilder builder =
			em.getCriteriaBuilder();

		// Этап 2. Создаем объект criteria
		CriteriaQuery<Letter> criteria =
				builder.createQuery(Letter.class);

		// Этап 3. Создаем объект root
		Root<Letter> root =
			criteria.from(Letter.class);

		// Этап 4. Преобразовываем объект criteria,
		// включая использование объекта root
		criteria.select(root);

		// Этап 5. Добавляем сортировку
		criteria.orderBy(builder.desc(root.get("date")));

		//Этап 6. Формируем запрос в виде объекта query
		TypedQuery<Letter> query = em.createQuery(criteria);

		//Этап 7. Получаем результат
		List<Letter> list =
				query.getResultList();

		//Этап 8. Обрабатываем результат
		return list;
	}

	// Получение объекта по ключу
	public Letter getLetter(int id) {
		Letter l =
			em.find(Letter.class, new Integer(id));

		return l;
	}

	// Добавить объект в базу данных
	public void addLetter(Letter letter) {
		em.persist(letter);
	}

	// Удалить объект из базы данных по ключу
	public void deleteLetter(int id) {
		Letter letter =
```

```
                em.find(Letter.class, new Integer(id));
                if(letter == null)
                        return;
        em.remove(letter);
    }

    // Модифицировать объект в базе данных
    public void modLetter(Letter letter) {
        if(letter == null)
                        return;
        Integer Id = letter.getId();

        Letter l =
                em.find(Letter.class, Id);
        if(l == null)
                        return;

        //System.out.println("Letters: модифицирую №" + Id);
        l.setDate(letter.getDate());
        l.setName(letter.getName());
        l.setText(letter.getText());
    }
}
```

Как видим, в класс ***Lets7*** и его интерфейс ***RemoteLets*** не внесено никаких существенных изменений, кроме смены имен и названия пакета.

Таким образом, завершена полная подготовка EJB-приложения ***Lets7*** для его преобразования в Web-сервис.

5.2.2 Аннотации поставщика Web-сервиса

Каждая Web-служба (Web-сервис) имеет определенную структуру.

Имеется набор общих требований, которым должна соответствовать каждая Web-служба [17]:

1) класс должен иметь аннотацию ***@javax.jws.WebService*** или XML-представление, описанное в дескрипторе развертывания ***webservices.xml***;
2) класс может реализовать нуль или более интерфейсов, которые должны иметь аннотацию ***@WebService***;
3) класс должен быть определен как ***public*** и не должен иметь спецификаторы ***final*** или ***abstract***;
4) класс должен иметь ***public***-конструктор по умолчанию;
5) класс не должен определять метод ***finalize()***;
6) для того чтобы преобразовать Web-службу SOAP в компонент-конечную точку, класс должен иметь аннотацию ***@javax.ejb.Stateless*** или ***@javax.ejb.Singleton***;

7) служба должна быть объектом, не сохраняющим состояние, и не должна сохранять характерное для клиента состояние во время вызовов методов.

Для того, чтобы превратить наше EJB-приложение в Web-службу SOAP, необходимо классу ***Lets7*** добавить аннотацию ***@WebService***.

Главной аннотацией Web-службы является ***@WebService***, общее описание которой представлено на листинге 5.6.

Листинг 5.6 — Определение аннотации @WebService

```
@Retention(RUNTIME) @Target(TYPE)
public @interface WebService {
            String name() default "";
            String targetNamespace() default "";
            String serviceName() default "";
            String portName() default "";
            String wsdlLocation() default "";
            String endpointInterface() default "";
}
```

Я не буду подробно описывать назначение каждого атрибута аннотации ***@WebService***. Для этого необходимо очень хорошо знать язык WSDL. Отмечу лишь два момента:

1) аннотацию можно использовать без переопределения атрибутов, тогда система будет использовать значения по умолчанию;
2) атрибут ***serviceName*** позволяет изменить имя Web-сервиса, если это — необходимо.

По умочанию, когда аннотация ***@WebService*** используется без параметров, имя сервиса для пользователя будет складываться из имени класса ***Lets7*** и слова ***Service***.

Добавив аннотацию к классу ***Lets7***, как показано на листинге 5.7, мы получим Web-сервис.

Листинг 5.7 — Преобразование EJB-компоненты Lets7.java в Web-сервис

```
@WebService
@Stateless
@LocalBean
public class Lets7 implements RemoteLets
{
    @PersistenceContext(name = "lab4-unit2")
    private EntityManager em;
    // Методы Web-сервиса ...
```

```
}
```

Чтобы наш Web-сервис стал доступным в сети, необходимо выполнить следующие действия:

а) запустить СУБД Apache Derby;

б) указать в дескрипторе ***web.xml*** класс ***Lets7*** как сервлет (см. листинг 5.8);

в) запустить сервер Apache TomEE.

Листинг 5.8 — Дескриптор развертывания проекта lab7 — файл web.xml

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

      <display-name>lab7</display-name>

      <!-- Указание класса сервлета -->
      <servlet>
            <servlet-name>lab7ws</servlet-name>
            <servlet-class>rsos.lab7.Lets7</servlet-class>
      </servlet>

      <!-- Указание каталога доступа к сервлету -->
      <servlet-mapping>
            <servlet-name>lab7ws</servlet-name>
            <url-pattern>/*</url-pattern>
      </servlet-mapping>

</web-app>
```

После того как сервер стартовал, следует проверить доступность Web-сервиса. Для этого нужно запустить браузер и выполнить запрос к серверу по адресу: http://localhost:8080/lab7/Lets7?wsdl.

В результате такого запроса, сервер ответит достаточно длинным и непонятным для непосвященных описанием сервиса на языке WSDL, как это показано на рисунке 5.3. По этой причине не рекомендуется делать дополнительные настройки сервиса, пока не изучен язык описания сервисов.

> При использовании только аннотации ***@WebService***, сервер сделает доступными все методы класса ***Lets7***, имеющими модификатор доступа ***public***.

С помощью атрибута ***operationName*** аннотации ***@WebMetod*** можно изменять отображаемое имя метода, а использование логического атрибута ***exclude*** делает метод отображаемым или нет. По умолчанию ***excluse = false***, что делает метод видимым.

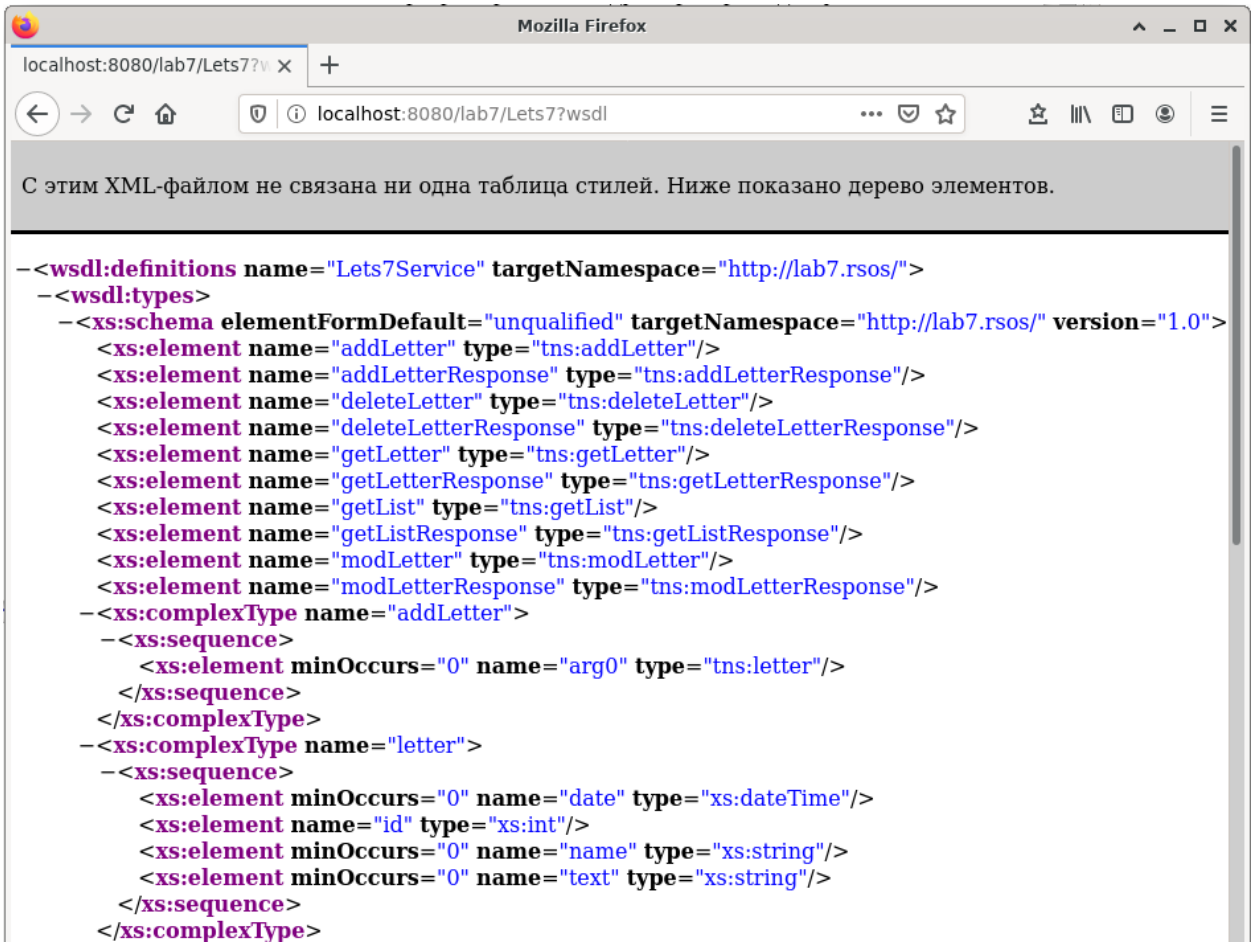

Рисунок 5.3 — Начало описания WSDL для Web-сервиса Lets7Service

Имеются и другие полезные аннотации, например, ***@OneWay*** следует использовать, если метод не возвращает значение. Тогда обращение выполняется асинхронно, что оптимизирует работу самого метода.

5.2.3 Обработка исключений поставщика Web-сервиса

Ошибки всегда имеются не только в программном обесечении поставщиков сервиса, но и в запросах клиентов сервисов.

В любом программном обеспечении всегда должен быть и имеется инструментарий для работы над ошибками. Такой инструментарий имеется и у поставщиков сервисов.

Естественно, что базовая проверка на ошибки осуществляется контейнерами сервлетов и EJB-компонент. Например, если в результате запроса клиета происходит обращение к несуществующему объекту, то на сервере будет генерироваться исключение ***NullPointerException***. В следствие этого, контейнер отправит клиенту специальный тип сообщения — ***SOAP Fault***.

Поставщик сервиса может создать свои собственные исключения, которые он может использовать, чтобы более точно характеризовать ошибки клиента сервиса. Для этого необходимо использовать аннотацию ***@WebFault***. Покажем использование таких исключений на примере класса ***Lets7***. Для этого, создадим и аннотируем класс ***Lets7Exception***, как показано на листинге 5.9.

Листинг 5.9 — Исходный текст класса Lets7Exception.java проекта lab7

```
package rsos.lab7;

import javax.xml.ws.WebFault;

@WebFault
public class Lets7Exception extends Exception
{
	private static final long serialVersionUID = 1L;

	// Пустой конструктор
	public Lets7Exception ()
	{
			super();
	}
	// Конструктор с сообщением
	public Lets7Exception (String message)
	{
			super(message);
	}
}
```

Обратите внимание, что второй конструктор этого класса имеет аргумент типа ***String***, с помощью которого поставщик сервиса может формировантъ специальные индивидуальные сообщения в ответах типа ***Fault***.

Допустим поставщик сервиса решил проверять в классе ***Lets7*** аргумент метода метода ***getLetter(int id)*** и формировать ответ ***Fault***, когда этот рагумент равен или меньше нуля. Тогда реализация этого метода может, например, быть такой, как показано на листинге 5.10.

Листинг 5.10 — Реализация метода getLetter(...) класса Lets7.java

```
// Получение объекта по ключу
public Letter getLetter(int id) throws Lets7Exception
{
	// Проверка аргумента с генерацией исключения
	if(id <= 0)
		throw new Lets7Exception(
				"Идентификатор объекта Letter не может быть <=0");
	Letter l =
			em.find(Letter.class, new Integer(id));

	return l;
}
```

Хотя этот пример во многом является надуманным, он правильно отображает технологию применения исключений на стороне поставщиков сервисов.

5.2.4 Обработка контекста Web-сервиса

Как и большинство из изученных нами технологий платформы Java EE, Web-службы SOAP имеют свой контекст среды. Доступ к этому контексту осуществляется с помощью ссылки на ***javax.xml.ws.WebServiceContext*** с аннотацией ***@Resource***.

Если мы хотим в классе ***Lets7*** использовать ссылку на контекст созданной Web-службы SOAP, то начальная часть этого класса должна выглядеть так, как это показано на листинге 5.11.

Листинг 5.11 — Обеспечение доступа к контексту среды в классе Lets7.java

```
package rsos.lab7;
import java.util.List;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import javax.xml.ws.WebServiceContext;

@WebService
@Stateless
@LocalBean
public class Lets7 implements RemoteLets
{
    // Доступ к контексту менеджера сущностей
    @PersistenceContext(name = "lab4-unit2")
    private EntityManager em;

    // Доступ к контексту среды Web-службы SOAP
    @Resource
    private WebServiceContext context;

    // Остальные определения методов класса Lets7
}
```

Объект контекста имеет множество разных методов. Наиболее важные из них представлены в таблице 5.3.

Таблица 5.3. - Методы объектов типа WebServiceContext [17]

Метод	***Описание метода***
getMessageContext	Возвращает MessageContext для запроса, который обслуживается на момент вызова метода. Он может быть использован для доступа к различным частям сообщения SOAP.
getUserPrincipal	Возвращает Principal, идентифицирующий отправителя запроса.
isUserInRole	Возвращает двоичное значение, указывающее, является ли аутентифицированный пользователь представителем определенной логической роли.
getEndpointReference	Возвращает EndpointReference, связанный с заданной конечной точкой.

В качестве примера использования контекста, потребуем, чтобы метод ***getDeleteLetter(...)*** мог применяться только пользователем с ролью ***Admin***, что соответственно показано на листинге 5.12.

Листинг 5.12 — Новая реализация метода getDeleteLetter() класса Lets7.java

```
// Удалить объект из базы данных по ключу
public void deleteLetter(int id) throws Lets7Exception
{
      // Проверка на наличие контекста
      if (context == null)
                  throw new Lets7Exception(
                        "Lets7: Контекст среды - недоступен!");

            // Проверка роли клиента
      if (!context.isUserInRole("Admin"))
                  throw new Lets7Exception(
                        "Только администратор может удалять письма!");

      // Проверка наличия аргумента вызова
      Letter letter =
            em.find(Letter.class, new Integer(id));
      if(letter == null)
                        return;

      // Непосредственное обращение на удаление объекта
      em.remove(letter);
}
```

Следует обратить внимание, что новая реализация метода не обрабатывает используемое исключение ***LetsException***, поэтому соответствующие изменения должны вноситься в интерфейс ***RemoteLets***.

Конечно, провайдер сервиса может использовать множество других технологических возможностей, предоставляемых сервисами контейнеров сервлетов и EJB-компонент. Мы ограничемся только приведенными примерами и перейдем к реализации потребителя сервиса.

5.3 Создание потребителя Web-службы SOAP

Теория предполагает, что потребитель сервиса читает его описание на языке WSDL как открытую книгу.

Конечно нормальный специалист, создающий или использующий Web-сервисы, должен уметь читать и понимать описания сделанные на языке WSDL. Проблема состоит в том, что в реальности эти описания являются достаточно объемными, а главное — содержат множество связей между своими частями. Если дополнительно учесть, что потребитель сервиса должен еще осознать практическую ценность самого сервиса, а также подразумевать изменения в обозначениях имен классов и методов, которые возможно сделал поставщик сервиса, то сама технология уже не становтся такой привлекательной, какой она казалась вначале изучения ее теоретических основ.

Причина сложности WSDL-описаний кроется не только в использовании базовой основы языка XML, но и в том, что само WSDL-описание является формально алгоримически полным, что не свойственно человеческому мышлению, которое всегда опирается на семантику контекста, уменьшая объем необходимых высказываний и их формальную точность.

Строгое формальное описание Web-сервиса на языке WSDL позволяет автоматизировать процесс созания программного обеспечения на стороне потребителя сервиса, что вполне вписывается в основные парадигмы программной платформы Java EE.

Прежде всего заметим, что поставщик сервиса не обязательно должен реализовывать сервис на основе платформы Java EE. Поставщик сервиса может использовать любую платформу, которая обеспечивает описание сервиса на языке WSDL. Таким образом, реализуя потребителя сервиса, даже для нашего примера, мы не должны считать, что сервис реализован именно так, как мы его описали в предыдущем пункте. Учитывая сказанное, последвательность изложения учебного материала данной главы представим в следующем порядке:

1) в пункте 5.3.1 рассмотрим аннотации для потребителя сервиса, которые предусмотреныплатформой Java EE;
2) в пункте 5.3.2 используем утилиту ***wsimport***, которая обеспечит нам создание базовых классов для написания потребителя сервиса;
3) в пункте 5.3.3 проведем реализацию тестового примера для потребителя сервиса;
4) в пункте 5.3.4 подведем итоги по результатам изучения данной главы.

5.3.1 Аннотации для потребителей сервиса

Потребитель Web-сервиса (***Service Consumer***) может быть реализован на программной платформе Java SE (***Java Standard Edition***).

В общем случае реализация Service Consuver может быть выполнена на любой плактформе, которая «*понимает*» язык WSDL, «*способна*» формировать SOAP-запросы и обрабатывать SOAP-ответы.

В данной дисциплине, мы воспользуемся средствами программной платформы Java, которая имеет утилиту ***wsimport***, создающую по описанию WSDL все необходимые файлы классов, причем как в исходных текстах, так и в бинарном виде. Но об этом — в следующем пункте.

Результаты доступа потребителя сервисов не предназначены для прямого вывода в браузер.

Потребителем сервиса может быть любое приложение, реализующее прокси-службу для осуществления удаленных запросов к Web-службе провайдера сервиса. В этом отношении, любой клиент сервиса аналогичен клиентам распределенных систем, реализуемых средствами объектных подходов, например, средствами технологии CORBA, изученной в бакалаврской дисциплине «*Распределенные вычислительные системы*». Потребитель Web-сервиса отличается лишь тем, что он вместо протокола **IIOP** (*Internet Inter-Orb Protocol*) или протокола RMI использует протокол SOAP.

Если потребитель сервиса функционирует в контейнерах клиентских приложений, сервлета или EJB платформы Java EE, то он может внедрять в свой код Web-службы с помошью аннотации ***@javax.xml.ws.WebServiceRef***. Например, если мы хотим реализовать потребителя сервиса в проекте ***lab7*** в виде класса ***ClientLetts***, то доступ к сервису ***Lets7Service*** будет выглядеть так, как показано на листинге 5.13.

Листинг 5.13 — Исходный текст класса Lets7Client.java проекта lab7

```
package rsos.lab7;
import javax.xml.ws.WebServiceRef;

public class ClientLets
{
	// Внедрение ссылки на Web-сервис в классе-потребителе сервиса
	@WebServiceRef
	private static Lets7 service;

	public static void main(String[] args)
	{
		// Реализация потребителя сервиса
```

```
	}
}
```

Если мы имеем много классов-потребителей сервиса, то можно создать вспомогательный класс, например, ***WebServiceProducer***, как это показано на листинге 5.14.

Листинг 5.14 — Вспомогательный класс-ссылка на Web-сервис проекта lab7

```
package rsos.lab7;
import javax.enterprise.inject.Produces;
import javax.xml.ws.WebServiceRef;

public class WebSeviceProducer
{
	@Produces
	@WebServiceRef
	private Lets7 service;
}
```

Тогда внедрение сервиса можно осуществлять как компоненту CDI, что показано на листинге 5.15.

Листинг 5.15 — Внедрение CDI в класс Lets7Client.java проекта lab7

```
package rsos.lab7;
import javax.inject.Inject;

public class ClientLets
{
	// Внедрение ссылки на Web-сервис в классе-потребителе сервиса
	@Inject
	private static Lets7 service;

	public static void main(String[] args)
	{
		// Реализация потребителя сервиса
	}
}
```

Основная идея реализации потребителя сервиса — обеспечение удаленного доступа к средствам поставщика сервиса.

Учебная цель данного подраздела — реализация удаленного доступа к поставщику сервиса, реализованного в проекте ***lab7***.

Исходя из поставленной цели, создадим новый проект ***lab8***, в котором откроем класс ***Lets7Client*** с ***package rsos.lab8***, а в следующих двух пунктах проведем демонстрацию тестового примера, реализующего доступ к сервису проекта ***lab7***.

5.3.2 Использование утилиты wsimport

Формальный подход к реализации тестового примера потребителя сервиса — использование утилиты ***wsimport***.

Создадим в домашней директории пользователя ***upk*** каталог ***lab8*** и поместим в него исполняемый сценарий ***ws_lab8.sh***, содержимое которого показано на рисунке 5.4.

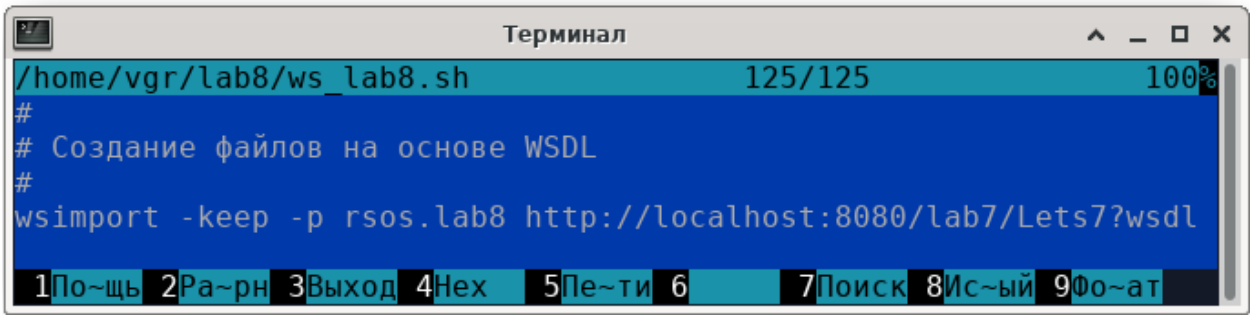

Рисунок 5.4 — Исходный текст сценария ws_lab8.sh

Приведенный сценарий запускает утилиту ***wsimport***, которая обращается по адресу доступа к WSDL-описанию сервиса ***Lets7***. Ключ ***-keep*** указывает, что результат работы утилиты будет помещен в текущую директорию, а ключ ***-p*** задает имя пакета генерируемых классов на языке Java. Более полное описание возможностей утилиты ***wsimport*** следует изучать по руководству ***man***.

Далее нужно:

1) запустить сервер СУБД Apache Derby;
2) в среде Eclipse EE запустить сервер приложений Apache TomEE;
3) в браузере установить соединение с сервером приложений по адресу http://localhost:8080/lab7/Lets7wsdl и убедиться, что он отображает описание Web-сервиса на языке WSDL;
4) в каталоге ***lab8*** запустить на исполнение сценарий ***ws_lab8.sh***.

После указанных действий к каталоге ***lab8*** появится дерево каталогов ***rsos/lab8***, где будет находиться множество файлов, как показано на рисунке 5.5.

В целом, утилита ***wsimport*** полностью реализует все классы, необходимые для реализации тестового приложения.

Проведем анализ полученного результата, внимательно рассмотрев содержимое файлов ***Lets7.java*** и ***Lets7Service.java***.

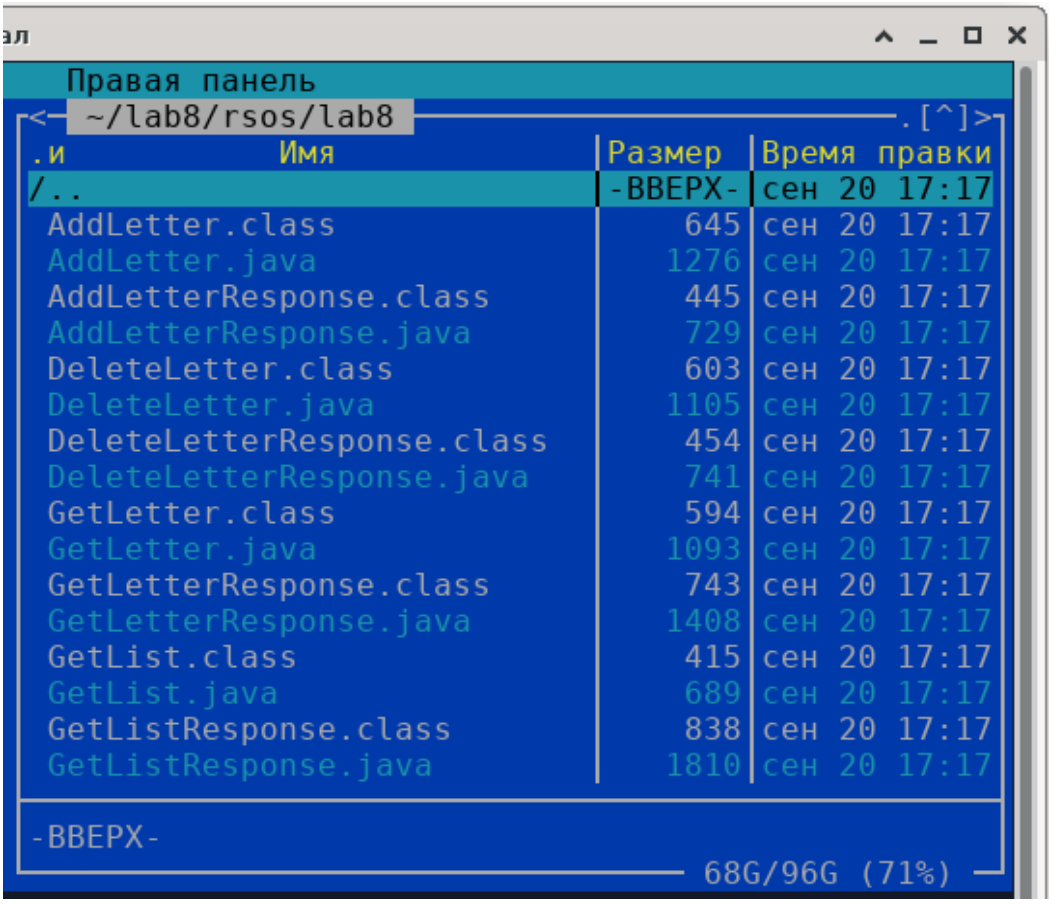

Рисунок 5.5 — Список файлов, сгенерированный утилитой wsimport

Файл ***Lets7.java*** является интерфейсом, в котором определены все необходимые удаленные методы, что подтверждается рисунком 5.6.

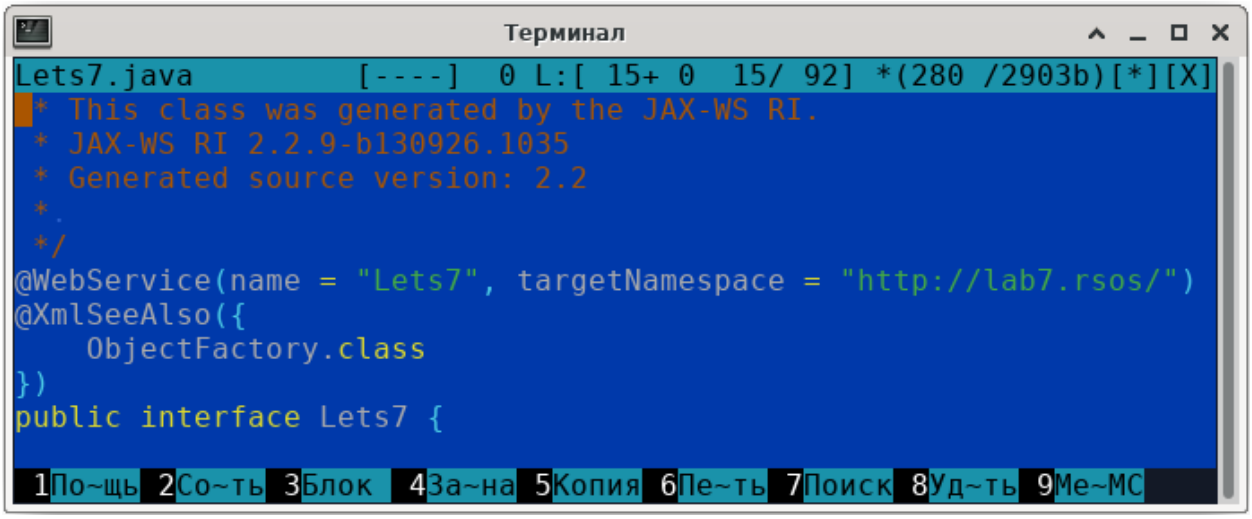

Рисунок 5.6 — Начало содержимого файла Lets7.java

Файл ***Lets7Service.java*** содержит несколько конструкторов для создания объектов типа ***Lets7Setvice***, что показано на листинге 5.16.

Листинг 5.16 — Исходный текст класса Lets7Service.java для проекта lab8

```
package rsos.lab8;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceFeature;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.2.9-b130926.1035
 * Generated source version: 2.2
 *
 */
@WebServiceClient(name = "Lets7Service",
      targetNamespace = "http://lab7.rsos/",
      wsdlLocation = "http://localhost:8080/lab7/Lets7?wsdl")
public class Lets7Service extends Service
{
    private final static URL LETS7SERVICE_WSDL_LOCATION;
    private final static WebServiceException LETS7SERVICE_EXCEPTION;
    private final static QName LETS7SERVICE_QNAME =
                  new QName("http://lab7.rsos/", "Lets7Service");

    static {
        URL url = null;
        WebServiceException e = null;
        try {
            url = new URL("http://localhost:8080/lab7/Lets7?wsdl");
        } catch (MalformedURLException ex) {
            e = new WebServiceException(ex);
        }
        LETS7SERVICE_WSDL_LOCATION = url;
        LETS7SERVICE_EXCEPTION = e;
    }

    public Lets7Service() {
        super(__getWsdlLocation(), LETS7SERVICE_QNAME);
    }

    public Lets7Service(WebServiceFeature... features) {
        super(__getWsdlLocation(), LETS7SERVICE_QNAME, features);
    }

    public Lets7Service(URL wsdlLocation) {
        super(wsdlLocation, LETS7SERVICE_QNAME);
    }
```

```
    public Lets7Service(URL wsdlLocation, WebServiceFeature... features) {
        super(wsdlLocation, LETS7SERVICE_QNAME, features);
    }

    public Lets7Service(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public Lets7Service(URL wsdlLocation, QName serviceName,
                WebServiceFeature... features) {
        super(wsdlLocation, serviceName, features);
    }

    /**
     *
     * @return
     *     returns Lets7
     */
    @WebEndpoint(name = "Lets7Port")
    public Lets7 getLets7Port() {
        return super.getPort(
                new QName("http://lab7.rsos/", "Lets7Port"), Lets7.class);
    }

    /**
     *
     * @param features
     *     A list of {@link javax.xml.ws.WebServiceFeature} to configure
     *     on the proxy.  Supported features not in the <code>features</code>
     *     parameter will have their default values.
     * @return
     *     returns Lets7
     */
    @WebEndpoint(name = "Lets7Port")
    public Lets7 getLets7Port(WebServiceFeature... features) {
        return super.getPort(
                new QName("http://lab7.rsos/", "Lets7Port"),
                      Lets7.class, features);
    }

    private static URL __getWsdlLocation() {
        if (LETS7SERVICE_EXCEPTION!= null) {
            throw LETS7SERVICE_EXCEPTION;
        }
        return LETS7SERVICE_WSDL_LOCATION;
    }
}
```

Обратите также внимание, что класс ***Lets7Service*** имеет два метода с именем ***getLets7Port()***. Оба метода возвращают объект типа ***Lets7***, который обеспечивает прокси-вызовы к удаленному объекту указанного типа. Таким образом, мы получили все необходимые средства для реализации тестового приложения, а теперь необходимо остановить сервер приложений и перенести все файлы с раширением ****.java*** в каталог проекта ***lab8***. В нашем случае, — это каталог с полным именем: ***/home/upk/rsos/lab8/src/rsos/lab8***.

5.3.3 Реализация тестового примера

После копирования файлов созданных утилитой ***wsimport*** в проект ***lab8***, следует сделать ***refrash*** этого проекта, чтобы среда разработки учла новые файлы.

План реализации тестового примера состоит из следующей последовательности шагов:

1) ***шаг 1*** — создаем объект типа ***Lets7Service***, а затем, с помощью его метода ***getLets7Port()*** получаем ссылку на удаленный объект типа ***Lets7***;
2) ***шаг 2*** — читаем список записей типа Letter и распечатываем его, убеждаясь в работоспособности технологии Web-служб SOAP;
3) ***шаг 3*** — читаем запись объекта типа ***Letter*** с идентификатором ***id=0*** и должны получить исключение с сообщением;
4) ***шаг 4*** — делаем запрос на удаление объекта типа ***Letter*** с идентификатором ***id=1*** и должны получить исключение с сообщением.

Тестовый пример реализуем в методе ***main(...)*** класса ***ClientLets***, как это показано на листинге 5.17.

Листинг 5.17 — Исходный текст класса ClientLets.java для проекта lab8

```
package rsos.lab8;

import java.io.PrintStream;
import java.util.List;

public class ClientLets
{
      public static void main(String[] args)
                  throws Lets7Exception_Exception
      {
        // Объект печати
        PrintStream out =
                        System.out;
        out.println("ClientLets: Начинаю работу...");

        // Получаем сетевой объект сервиса
        Lets7Service service =
                  new Lets7Service();
        // Получаем прокси на удаленный объект
        Lets7 remote =
                  (Lets7) service.getLets7Port();

        // Проверяем, что ссылка получена
        if(remote == null)
        {
            out.println(
                        "Нет соединения с удаленным объектом...");
            return;
        }
        // Проверяем чтение списка записей типа Letter
```

```
        List<Letter> list =
                remote.getList();
        if(list.size() <= 0)
        {
            out.println("Получили пустой список...");
            return;
        }
        out.println("Список объектов Letter:");

        String[] ss;
        // Цикл по списку объектов
        for(Letter l : list)
        {
            ss = l.date.toString().split("T");

            out.println("[" + l.id
                        + " " + ss[0] + " " + ss[1].substring(0, 8)
                        + " " + l.name
                        + " " + l.text.trim() + "]");
        }

        Letter l;
        // Пытаемся получить объект Letter с id=0
        try {
            l =
                remote.getLetter(0);
            out.println(l.toString());
        }catch(Lets7Exception_Exception e) {
                out.println("Получил исключение:"
                      + e.getLocalizedMessage() + "\n"
                      + e.toString() + "\n");
        }

        // Пытаемся удалить объект Letter с id=1
        try {
                remote.deleteLetter(0);
        }catch(Lets7Exception_Exception e) {
                out.println("Получил исключение:"
                      + e.getLocalizedMessage() + "\n"
                      + e.toString() + "\n");
        }
    }
}
```

Теперь запустим сервер приложений, а затем — класс ***ClientLets*** на исполнение. После этих действий, мы получим результат, показанный на рисунке 5.7.

По результатам проведенного теста можно сделать главный вывод, что программная платформа Java EE позволяет успешно и с минимальными интеллектуальными затратами реализовывать Web-службы SOAP.

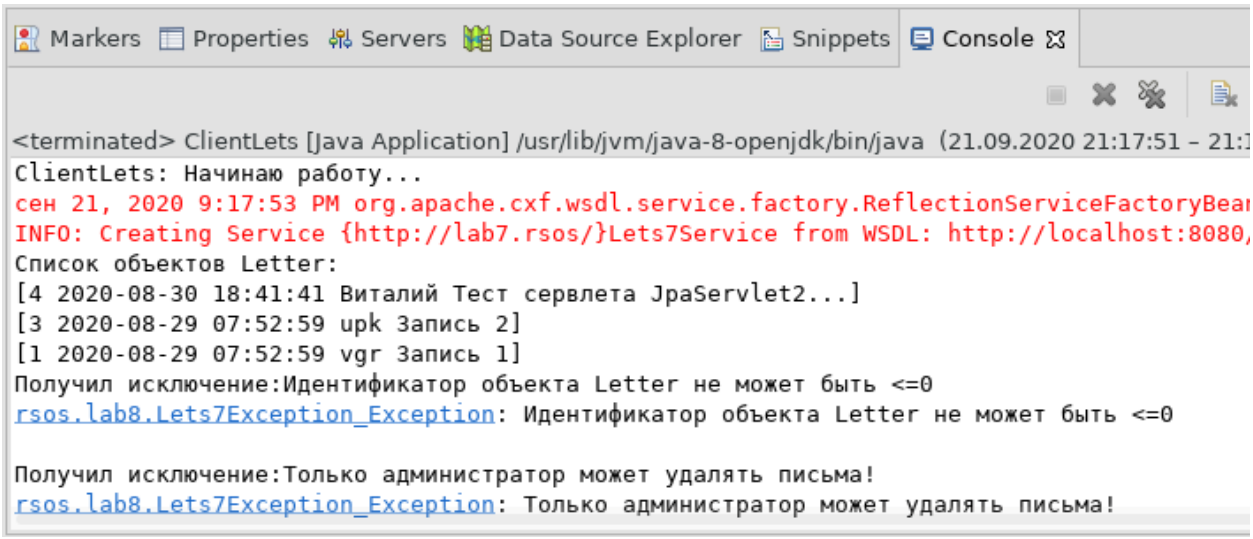

Рисунок 5.7 — Результат запуска тестового приложения ClientLets

5.3.4 Выводы по результатам изучения главы 5

Технология Web-служб на основе протокола SOAP обеспечивает достижение основной цели дисциплины «Распределенные сервис-ориентированные системы» — оперативное решение задач масштаба предприятий, возникающих на уровне бизнес-логики.

Не вдаваясь в расширительные толкования понятия сервис-ориентированных систем, учебный материал данной главы наглядно показывает, что инструментальные средства программной платформы Java EE обеспечивают:

- разделение масштабных прикладных задач минимум на две части, реализуемые различными группами, названными как ***поставщики сервиса*** и ***потребители сервиса***;
- возможность реализации решения масштабных задач поставщиками сервиса ***с использованием служебного сервиса***, предоставляемого кон-тейнерами серверов приложений, а также — возможности публикации универсального описания сервиса, доступного их потребителям;
- возможность использования опубликованного сервиса ***для оперативной реализации приложений*** потребителями сервиса.

Программная платформа Java EE, применительно к технологии Web-служб на основе протокола SOAP, фактически напрямую обеспечивает реализацию бизнес-парадигмы архитектуры предприятия, рассмотренной в пункте 1.3.4 первой главы данного учебного пособия.

Действительно, программная платформа Java EE позволяет поставщику сервиса разрабатывать различные приложения, используя возможности служеб-

ных сервисов серверов приложений. Такие разработки ведутся IT-специалистами из потребностей решения конкретных задач, используя все возможности объектно-ориентированных языков и инструментальных средств баз данных (СУБД). На основе разработанных приложений можно создавать EJB-компоненты, которые сами по себе могут быть распределенными приложениями. Все эти работы могут производиться без прямого согласования с потребителями сервисов, опираясь на техническую базу, которая имеется у поставщика сервиса.

Когда потребтели сервиса, соответствующие уровню бизнес-логики архитектуры предприятия начинают формировать свои потребности, тогда поставщики сервиса могут достаточно быстро реализовать необходимые интерфейсы своих приложений и согласовать их на должном уровне, что соответствует уровню интерфейса сервиса (см. рисунок 1.12 главы 1, стр. 33).

Окончательно, IT-специалисты могут достаточно быстро реализовать и опубликовать WSDL-описания своих сервисов, используя соответствующие аннотации кратко описанные в подразделе 5.1 данной главы. Также они могут воспользоваться утилитой ***wsgen***, упомянутой в пункте 5.1.5 этой же главы. Далее, на основе WSDL-описаний необходимых сервисов, с помошью утилиты ***wsimport*** может быть сгенерирован необходимый софт, на основе которого быстро могут быть реализованы интерфейсные программы для потребителей сервисов. В частности, в качестве инструмента для реализации приложений, может быть использована технология JSF, подробно описанная во второй главе данного пособия.

По результатам изучения материала пятой главы можно сделать следующие выводы:

1) в подразделе 5.1 изучены теоретические основы построения сервис-ориентированных систем;
2) в подразделе 5.2 представлен практический пример реализации серверной части Web-сервиса, которую осуществляет поставщик сервиса.
3) в подразделе 5.3 представлен тестовый пример использования сервиса, который реализуется его потребителем;
4) открытыми остались вопросы обеспечения безопастности разработки и использования Web-сервисов, что в общем случае выходит за рамки учебного материала данной дисциплины.

Тематика безопастности использования программного обеспечения сервис-ориентированных систем решается различными способами. В частности — средствами использования защищенных каналов связи или написаниянаписания отдельных приложений, которые сами являются Web-сервисами.

Вопросы для самопроверки

1. Изобразите графически общую архитектуру взаимодействия составляющих сервис-ориентированных систем.
2. Когда были сформированы основные принципы технология Web-сервисов первого поколения?
3. Какова общая тенденция развития современных Web-технологий?
4. В чем состоит основная суть и назначение языка WSDL?
5. Какие особенности протокола SOAP вы знаете?
6. В чем состоит назначение и перспективы использования технологии UDDI?
7. В чем состоит назначение основных элементов и атрибутов языка WSDL?
8. Каковы основные структурные элементы сообщений протокола SOAP?
9. Каково назначение сообщения Fault?
10. Какие утилиты платформы Java SE обеспечивают реализацию Web-сервисов на основе протокола SOAP?
11. Какие аннотации платформы Java EE может использовать поставщик сервисов?
12. Какому набору требований должна удовлетворять Web-служба?
13. Какой дескриптор развертывания может заменить аннотации, предназначенные для Web-служб?
14. Каким образом потребитель сервиса узнает о возможностях поставщика сервиса?
15. Каким образом потребитель сервиса узнает об исключениях, возникщих в программном обеспечении поставщика сервиса?
16. Какими средствами обеспечивается доступ к контексту Web-службы?
17. Какими аннотациями может воспользоваться потребитель сервиса?
18. Для чего предназначена аннотация WebServiceProducer?
19. Какие условия необходимы потребителю сервиса для использования аннотации @Inject?
20. Что обеспечивает утилита wsimport для потребителя сервиса?

6 Тема 6. Web-службы в стиле REST

Технология REST является современной популярной альтернативой Web-службам SOAP.

Любая альтернатива, включая распределенные сетевые технологии, основана на недостатках объектов и включающих их систем, которые мы изучаем или используем в практической деятельности.

Технология REST (***Re****presentational* ***S****tate* ***T****ransfer*) — технология передачи состояния представления, означающая, что REST-запрос клиента к серверу содержит всю нужную информацию о желаемом ответе сервера.

Основной недостаток модели SOA на основе протокола SOAP — необходимость описания взаимодействия компонентов распределенной системы на языке WSDL, которое для многих приложений является явно избыточным.

В основе языка WSDL лежит язык XML, обеспечивающий не только описание самого взаимодействия распределенных систем, но также — описание XML-схем, необходимых для синтаксического контроля таких описаний. В результате, проектные решения на основе протокола SOAP становятся сложными не только для поставщиков сервисов, но главное — это является сложным для основной массы потребителей этих сервисов.

Передача состояния представления (REST) — архитектурный стиль проектирования и реализации распределенных сервис-ориентированных систем, максимально использующих возможности Web-технологий.

Основная позитивная идея сторонников REST-технологий — замена сложных описаний сетевого взаимодействия на языке WSDL на URI/URN-ресурсы, представленные (опубликованные) в контенте уже используемых средств гипермедиа, например, в тексте обычных страниц на языке HTML. Таким образом, потребитель сервиса, вместо чтения описаний интерфесов и реализации агентов, обеспечивающих доступ к сервисам через интерфейсы, просто выбирает ссылку в окне браузера и получает необходимый сервис.

Учебная цель данной главы — краткое изучение REST-технологий применительно к предметной области изучаемой дисциплины — распределенным сервис-ориентированным системам. Изучаемая тематика раскрывается в трех подразделах:

1) подраздел 6.1 — раскрывает базовые положения технологии RESTful;
2) подраздел 6.2 — показывает пример реализации Web-службы;

3) подраздел 6.3 — показывает возможности потребителей сервиса.

6.1 Основные положения технологии RESTful

Web-службы, построенные с учетом ограничений REST-технологий, называются RESTful-системами.

В отличие от Web-сервисов, построенных на основе протокола SOAP, RESTful-системы не имеют официального стандарта на свое API, поэтому принято считать, что — это архитектурный стиль программирования.

RESTful — это архитектурный стиль проектирования и реализации сервис-ориентированных систем.

Считается, что во многом идейной парадигмой технологии REST стала стандартная классификация действий по работе с базами данных CRUD, которая была введена Джеймсом Мартином в 1983 году.

CRUD — это акроним, обозначающий четыре базовых действия (функций) для работы с базами данных:

1) CREATE — создание записей (*INSERT*);
2) READ — чтение записей (*SELECT*);
3) UPDATE — модификация записей;
4) DELETE — удаление записей.

Применительно к Web-службам, термин CRUD проецируется на четыре HTTP-запроса к Web-серверам:

1) POST — запрос на ***создание*** ресурса;
2) GET — запрос на ***получение*** ресурса;
3) PUT — запрос на ***модификацию*** ресурса;
4) DELETE — запрос на ***удаление*** ресурса.

Сам термин REST был введен Роем Филдингом в 2000 году и упоминается в ключе его докторской диссертации: «*Архитектурные стили и дизайн сетевых программных архитектур*». Им было предложено и шесть ограничений, нарушение которых не позволяет считать сервис-приложение REST-системой:

1. **Модель «Клиент-сервер»** — предполагается приведение приложения, возможно целостного по начальной реализации, к архитектуре, где выделены поставщик сервиса и потребители сервиса. Это улучшает мастабируемость приложения и позволяет отдельным частям развиваться независимо друг от друга.
2. **Отсутствие состояния** — обеспчение на стороне сервера протокола взаи-

модействия без сохранения состояния. При этом, состояние сессии должно сохраняться на стороне клиента (потребителя сервиса).

3. **Кэширование ответов сервера** — способность сервера или промежуточных узлов кэшировать свои ответы. Это позволяет повысить производительность и расширяемость системы.
4. **Единообразие интерфейсов сервисов** — фундаментальное требование дизайна REST-сервисов. Унификация интерфесов должна соответствовать четырем дополнительным условиям: идентификации ресурсов посредством URI; возможности манипуляции ресурсов на основе представлений; использованию «*самозаписываемых*» сообщений и применению гипермедиа как средства изменения состояния приложения.
5. **Слои взаимодействия** — использование взаимодействия клиента и сервера на основе иерархической структуры сетей.
6. **Код по требованию** (*необязательное ограничение*) — возможность расширения функциональности клиентов за счет загрузки кода с серверов приложений в виде апплетов или сценариев.

По мнению Роя Филдинга, приложения, не соответствующие приведенным условиям, не могут называться REST-приложениями.

По мнению Филдинга, приложения, которые соответствуют указанным выше шести условиям, получают следующие преимущества:

а) ***надежность*** по причине отсутствия необходимости сохранять информацию о состоянии клиента, поскольку она может быть утеряна;

б) ***производительность*** за счет использования кэша и ***масштабируемость*** за счет разделения поставщиков и потребителей сервисов;

в) ***простота интерфейсов*** и ***портативность компонентов*** создаваемых сервисных систем;

г) ***легкость внесения изменений*** и ***способность «эволюционировать»*** под воздействием новых требований к системам.

Основной недостаток REST-технологий — отсутствие формальной стандартизации, четко ограничивающей их объективное присутствие.

Имеющиеся недостатки REST-технологий являются продолжением их преимуществ. Действительно, REST-технологии полностью ориентированы на использование только Web-технологий, что в конечном итоге приводит к существенному развитию последних, но ограничивает первых.

Чтобы не впадать в крайности популистских утверждений, рассмотрим заявленную тематику данного подраздела только в следующих трех аспектах:

1) общие понятия и обозначения изучаемой тематики в планах ресурсов, адресации и других представлений, используемых в теоретических описаниях технологий REST;
2) возможности протокола HTTP и перспективы использования языка WADL;
3) описание технологии JAX-RS, как основного инструментального средства реализации конкретных REST-систем посредством программной платформы Java EE.

6.1.1 Ресурсы, URI, представления и адресуемость

Базовым понятием Web-технологий является понятие ресурса.

REST-технологии можно рассматривать как дальнейшее развитие Web-технологий применительно к распределенным сервис-ориентированным системам. Соответственно, в REST-технологиях, в явном или неявном виде, присутствуют такие понятия как ресурсы, URI/URN, представления и адресуемость.

Ресурс — это зонтичный термин, обозначающий любую сущность, на которую клиент (потребитель сервиса) может поставить ссылку или с которой он пытается или может попытаться взаимодействовать. В такой интерпретации, ресурсы могут обозначать что угодно: файлы в файловой системе, базу данных, сайт или любую информацию, которая попала в поле зрения пользователя.

URI (*Uniform Resource Identifier*) — унифицированный или единообразный идентификатор ресурса, имеющий две базовые интерпретации:

1) **URL** (*Uniform Resource Locator*) — система унифицированных адресов электронных ресурсов, которая в современном понимании соответствует понятию URI;
2) **URN** (*Uniform Resource Name*) — единообразное название или имя ресурса, которые не включают в себя указания на местонахождение и способ обращения к ресурсу.

Что касается тематики REST-технологий, то под URI понимается именно URL, указывающий на местоположение ресурса в формате:

http://host:port/path?queryString#fragment, где

http — протокол соединения с ресурсом;

host — это DNS-имя или IP-адрес ресурса;

port — необязательная часть формата, указывающая порт соединения примени-

тельно к стеку протоколов TCP/IP;

path — это относительный адрес ресурса в пределах файловой системы сервера, использующий разделительный знак «/»;

queryString — необязательной список параметров, представленных в форме «***имя=значение***», где разделительным символом между такими парами является знак «**&**»;

fragment — это указатель на конкретное место в документе.

Указанный формат URI соответствует методу ***GET*** протокола HTTP, используемого в классических интерпретациях Web-технологий.

Обратите внимание, что это всего лишь один из методов протокола HTTP, который способен полностью отображаться в адресной строке браузеров. Это создает впечатление простоты понимания самой адресации, что пропагандируется Роем Филдингом, но реально применяется только в простейших запросах и то — на английском языке.

С другой стороны, сам адрес не обязательно нужно отображать на HTML-страницах, читаемых пользователями сервисов. Этот адрес может быть скрыт и заменен подходящими именами, характеризующими ресурс на национальных языках пользователей Интернет. Такой подход широко используется на практике, что подчеркивает хорошую адаптируемость стиля REST.

Представления — это характеристики форматов тех сущностей, которые адресуются с помощью URI.

Компьютерные технологии, включая технологии WEB, разработали множество форматов представления информации. В настоящее время, общепринятым способом обозначения различных представлений являются MIME-типы.

MIME (***M**ultipurpose **I**nternet **M**ail **E**xtensions*) — это многоцелевые расширения интернет-почты, ставшие стандартом, описывающим спецификации для кодирования информации и форматирования сообщений. Базовый набор таких спецификаций определен в ряде документов: ***RFC 2045***, ***RFC 2046***, ***RFC 4288***, ***RFC 4289*** и ***RFC 4855***.

Официальный список всех MIME-типов публикуется Администрацией адресного пространства Интернет, сокращенно обозначаемого **IANA** (*Internet Assigned Numbers Authority*).

Адресуемость — основной принцип проектирования Web-служб в стиле RESTful.

Идея адресуемости — достаточно проста. Если меется сетевой объект, с которым работает потребитель сервиса, то такая сущность должна иметь кон-

кретный сетевой адрес. Более того, поскольку потребитель сервиса получает и работает с HTML-страницей, то на ней должны быть указаны адреса функций, которые обрабатывают указанный сетевой объект. Такой подход должен минимизировать объем информации, вводимой пользователем сервиса, что в конечном итоге не только упрощает работу с сервисом, но и уменьшает количество ошибок, возникающих по вине пользователя.

В качестве примера, Рой Филдинг указывает на так называемый «сетевой *серфинг*», когда нужная информация находится пользователем посредством нескольких «*кликов*» устройством мыши. Очвидно, что для приложений, которые требуют иных действий, технология REST не дает столь убедительного эффекта.

6.1.2 Протокол HTTP

В отличие от технологии SOAP, RESTful ориентирован на использование только протокола HTTP.

Технология RESTful ориентируется на максимальное использование возможностей протокола HTTP, а также — на непосредственное представление информации в браузерах пользователей.

Как было показано в начале текущей главы, идейная основа CRUD, зародившаяся в недрах централизованного использования баз данных, управляемых СУБД, легко проецируется на четыре метода доступа к Web-серверам по протоколу HTTP. Это — методы: POST, GET, PUT и DELETE. И хотя большинство браузеров поддерживают только методы GET и POST, сами Web-технологии допускают использование еще четырех типов запросов:

1) запрос ***HEAD*** — идентичен запросу GET, но не возвращает в ответ тело сообщения;
2) запрос ***TRACE*** — отражает полученный запрос назад клиенту, позволяя проверять, какую информацию добавляет к запросу или изменяет промежуточный сервер, прокси-сервер или брандмауэр;
3) запрос ***OPTIONS*** — выполняет запрос информации о возможностях коммуникации для той цепочки запросов/ответов, на которую указывает данный URI;
4) запрос ***CONNECT*** — используется с прокси-серверами для динамического переключения на работу в режиме туннеля.

Язык HTML и браузеры обеспечивают запросы только в виде методов GET и POST.

Возможности браузеров и языка HTML существенно ограничивают пря-

мое использование браузеров в реализации технологии REST. Возможно в будущем эта ситуация изменится, но в настоящее время полное применение стиля REST возможно только через ***Proxy***-серверы или серверы приложений. Таким образом, технология RESTful — гораздо шире узких ограничений языка HTML и протокол HTTP предоставляет ей дополнительные свойства, которые уже реализованы или будут реализованы в современных браузерах. Рассмотрим указанные возможности.

Согласование содержимого — это механизм автоматического определения необходимого представления при наличии нескольких разнотипных вариантов, использующий заголовки HTTP-запросов Accept, Accept-Charset, Accept-Encoding, Accept-Language и User-Agent, как это показано в таблице 6.1.

Таблица 6.1 — Базовый набор значений заголовков [17]

Имя заголовка	***Описание***
Accept	Допустимые типы содержимого, например, text/plain.
Accept-Charset	Допустимые наборы символов, например, utf-8.
Accept-Encoding	Допустимые варианты кодировки, например, gzip, deflate.
Accept-Language	Допустимый язык отклика, например, en-US или ru-RU.
Cookie	Файл HTTP-cookie, ранее отосланный сервером.
Content-Length	Длина тела запроса в байтах.
Content-Type	MIME-тип тела запроса, например, text/xml.
Date	Дата и время отправки сообщения.
ETag	Идентификатор конкретной версии ресурса, например, значение: 8af7ad3082f20958.
If-Match	Действие производится лишь в том случае, если клиент предоставил объект, совпадающий с таким же объектом на сервере.
If-Modified-Since	Допускает возврат кода состояния, например, если контент не изменялся с указанной даты, то: 304 — Не изменялось.
User-Agent	Строка, соответствующая конкретной реализации пользовательского агента, например, Mozilla/5.0.

Естественно, что потребность в использовании этих возможностей зависит от прикладного назначения самих приложений и навыков разработчика.

Типы содержимого — это MIME-типы, записываемые в полях заголовков ***Content-Type*** и ***Accept***, например:

а) ***text/plain*** — используется по умолчанию, им записываются простые текстовые сообщения;

б) ***text/html*** — базовый тип, применяемый в браузерах и информирующий пользовательский агент о получении Web-страницы на языке HTML;

в) ***image/gif***, ***image/jpeg***, ***image/png*** — обобщающий тип, соответствующий

изображениям нескольких типов и требующий соответствующего устройства отображения для просмотра информации;

г) ***text/xml***, ***application/xml*** — формат, используемый для обмена XML-сообщениями;

д) ***application/json*** — «*легковесный*» текстовый формат для обмена данными, не зависящий от конкретного языка программирования.

Безусловно, количество возможных MIME-типов — значительно шире, что требует изучения документов RFC, указанных в предыдущем пункте.

Важнейшей частью протокола HTTP являются возвращаемые сервером коды состояния.

Коды состояния — это трехзначные целые числа, которые описывают контекст пришедшего ответа. Первая цифра кода состояния указывает на один из классов ответа:

а) ***1xx — информационный***: запрос получен, процесс продолжает работу;

б) ***2xx — успех***: действие было успешно получено, интерпретировано и принято;

в) ***3xx — перенаправление***: необходимо выполнить дополнительные действия для полного удовлетворения запроса;

г) ***4xx — клиентская ошибка***: запрос содержит ошибочный синтаксис или не может быть удовлетворен;

д) ***5xx — серверная ошибка***: серверу не удалось выполнить запрос.

Полный список кодов возврата — достаточно большой. Его можно найти в источнике [27].

Прямой стиль адресации Web-служб — это стиль адресации ресурса, который позволяет обеспечивать над ним различные операции из разряда стандартного набора CRUD.

Поясним данное положение штатным примером нашей дисциплины, который демонстрирует сервис для работы со списком записей ***Letter***. С помощью стандартного набора CRUD мы можем: получить весь список записей, получить отдельную запись по ее номеру, а также — добавить новую запись, модифицировать или удалить любую запись по ее номеру.

Теперь, если сервис реализован в проекте ***lab9*** на хосте ***localhost*** и порту ***8080***, то адрес http://localhost:8080/lab9/letter можно понимать в двух вариантах:

1) для метода **GET** — прочитать все записи;

2) для метода **POST** — добавить новую запись.

Если к указанному адресу добавить целое число, которое интерпретиро-

вать как номер записи, например, http://localhost:8080/lab9/letter/72, то добавляются следующие три интерпретации:

1) для метода **GET** — прочитать запись с номером 72;
2) для метода **PUT** — модифицировать запись с номером 72;
3) для метода **DELETE** — удалить запись с номером 72.

Естественно, во многих случаях необходимо передавать дополнительные параметры запроса, поэтому прямой стиль адресации является далеко не универсальным. Тем не менее, для многих простых случаев — этого вполне достаточно.

6.1.3 Языки WADL и HAL

В августе 2009, компания Sun Microsystems предложила для моделирования ресурсов Web-сервисов язык WADL.

WADL (*Web Application Description Language*) — это машинно-читаемое XML-описание для Web-приложений, которое — подобно языку WSDL для SOAP, но использует только протокол HTTP.

Язык был предложен Всемирному консорциуму W3C, но не получил широкого распространения, возможно по причине не очень большой любви к языку XML. В любом случае, он еще не стандартизирован и W3C не имеет на него никаких планов.

Как отмечено ранее, REST является архитектурным стилем разработки Web-приложений, что накладывает на проектные решения дополнительные ограничения. Одним из таких архитектурных ограничений на REST-приложения является проект HATEOAS.

HATEOAS — это ограничение, требующее чтобы сервер обеспечивал взаимодействие с клиентом посредством динамического доступа через гипермедиа, что означает доступ к ресурсам через ***гиперссылки в гипертексте***.

> В настоящее время не существует универсального формата для предоставления ссылок между ресурсами.

Не вдаваясь в детальный анализ представленного выше утверждения, отметим, что существует два популярных формата представления ссылок в REST гипермедиа сервисах:

а) **RFC 5988** — документ, описывающий спецификации типов Web-ссылок [28];
б) **JSON Hypermedia API Language** — описание языка HAL, использующего гиперссылки в формате представления JSON [29].

6.1.4 Технология JAX-RS

Программная платформа Java EE обеспечивает полную поддержку технологии RESTful с помошью инструментальных средств прОекта JAX-RS.

JAX-RS (*Java API for RESTful Web Services*) — это спецификация API языка программирования Java, обеспечивающая поддержку создания Web-сервисов в соответствии с архитектурным шаблоном передачи состояния представления REST. Не имеет своего RFC, но содержит достаточно полное описание для версии 2.0 от 22 мая 2013 года, представленное корпорацией Oracle. Его можно найти в источнике [30], а полный перечень пакетов технологи JAX-RS, включающий как серверную, так и клиентскую части, представлен в таблице 6.2.

Таблица 6.2 — Основные пакеты JAX-RS [17]

Пакет	***Описание пакета***
javax.ws.rs	Высокоуровневые интерфейсы и аннотации, используемые для создания Web-служб с передачей состояния представления.
javax.ws.rs.client	Классы и интерфейсы клиентского API JAX-RS.
javax.ws.rs.container	API JAX-RS контейнера.
javax.ws.rs.core	Низкоуровневые интерфейсы и аннотации, используемые для создания Web-ресурсов с передачей состояния представления.
javax.ws.rs.ext	API, предоставляющие расширения для типов, поддерживаемых в JAX-RS API.

Все спецификации JAX-RS основаны только на аннотациях и не требуют дескриторов развертывания в формате XML-файлов.

RESTful-сервис реализуется специальным сервлетом, который представляет собой обычный JAVA-класс, аннотированный аннотацией ***@Path(...)*** и одной из дополнительных аннотаций ***@Stateless*** или ***@Singleton***.

RESTful-сервис использует достаточно много аннотаций, определенных в пакете ***javax.ws.rs***. Наиболее важные из них, влияющие на общую архитектуру реализации сервиса, представлены в таблице 6.3.

Аннотации ***@GET***, ***@PUP***, ***@POST***, ***@DELETE*** и ***@HEAD*** указывают допустимые типы запросов к RESTful-сервису, а аннотации ***@Produces(…)*** и ***@Consumes(…)*** — указывают типы ответов и запросов, представленные в виде констант их аргументов.

Медиатипы запросов и ответов определены в соответствующем классе ***javax.ws.rs.core.MediaType***, а их значения представлены в таблице 6.4.

Таблица 6.3 — Основные пакеты JAX-RS [17]

Аннотация	***Описание***
@Path(...)	Указывает относительный путь для класса или метода ресурса.
@GET, @PUP, @POST, @DELETE и @HEAD	Указывают тип HTTP-запроса, разделяя ПО сервиса на соответствующие части.
@Produces(...)	Указывает тип ответа (см. таблицу 6.4).
@Consumes(...)	Указывает принимаемый тип запроса (см. таблицу 6.4).

Таблица 6.4 — MIME-типы класса MediaType [17]

Имя константы	***MIME-тип***
APPLICATION_ATOM_XML	"application/atom+xml"
APPLICATION_FORM_URLENCODED	"application/x-www-form-urlencoded"
APPLICATION_JSON	"application/json"
APPLICATION_OCTET_STREAM	"application/octet-stream"
APPLICATION_SVG_XML	"application/svg+xml"
APPLICATION_XHTML_XML	"application/xhtml+xml"
APPLICATION_XML	"application/xml"
MULTIPART_FORM_DATA	"multipart/form-data"
TEXT_HTML	"text/html"
TEXT_PLAIN	"text/plain"
TEXT_XML	"text/xml"
WILDCARD	"*/*"

Главной аннотацией, по которой контейнер определяет тип RESTful-сервлета, является аннотация ***@Path(...)***, имеющая общий формат:

@Path("/Путь1/Путь2/.../ПутьN")

где ***ПутьK*** — слово, задающее часть пути и соответсвующие части ***path***, определенные ранее в адресе URI. Это слово также может быть представлено регулярным выражением, которое контролируется контейнером сервиса, в формате:

"{" variable-name [":" regular-expression] "}"

Чтобы продемонстрировать сказанное более наглядно, создадим в среде

Eclipse EE проект типа ***Dynamic Web Project*** с именем ***lab9***, не забыв указать создание файла ***web.xml***.

В указанном проекте создадим класс ***LetsRestService***, который будем рассматривать как шаблон, демонстрирующий работу со списком записей типа ***Letter***, рассмотренных ранее в предыдущих главах.

С учетом изученных аннотаций и условий отображения результатов сервиса в приложении браузера, указанный шаблон может быть представлен, как показано на листинге 6.1.

Листинг 6.1 — Исходный текст класса LetsRestService.java для проекта lab9

```
package rsos.lab9;
import javax.ejb.Stateless;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
 * Аннотации, определяющие сервлет типа RESTful, который
 * доступен по адресу: http://localhost:8080/lab9
 */
@Path("/")
@Stateless
public class LetsRestService
{
	// Простой тест работоспособности REST-сервиса
	/**
	 * Простейшая демонстрация доступа по методу GET, доступная
	 * по адресу: http://localhost:8080/lab9/test
	 */
	@GET
	@Path("/test")
	@Produces(MediaType.TEXT_PLAIN)
	public String testRS()
	{
		return "Проверка технологии JAX-RS";
	}
	// Получение списка записей
	/**
	 * Доступ по методу GET, читающий весь список записе, при
	 * обращении по адресу: http://localhost:8080/lab9/letter
	 */
	@GET
	@Path("/letter")
	@Produces(MediaType.TEXT_PLAIN)
	public String getLets()
	{
			return "Получение списка записей - не реализовано!";
	}
	// Получение записи по идентификатору id=1
```

```
	/**
	 * Доступ по методу GET, читающий только одну запись, при
	 * обращении по адресу: http://localhost:8080/lab9/letter/1
	 * Должен использоваться один параметр типа int.
	 */
	@GET
	@Path("/letter/1")
	@Produces(MediaType.TEXT_PLAIN)
	public String getLetter()
	{
		return "Получение записи по идентификатору - не реализовано!";
	}
	// Добавление новой записи
	@POST
	@Path("/add")
	@Produces(MediaType.TEXT_PLAIN)
	public String addLetter()
	{
			return "Добавление новой записи - не реализовано!";
	}
	// Удаление записи по идентификатору
	@POST
	@Path("/delete")
	@Produces(MediaType.TEXT_PLAIN)
	public String deleteLetter()
	{
		return "Удаление записи по идентификатору - не реализовано!";
	}
	// Модификация записи по идентификатору
	@POST
	@Path("/mod")
	@Produces(MediaType.TEXT_PLAIN)
	public String modLetter()
	{
		return "Модификация записи по идентификатору - не реализовано!";
	}
}
```

Следует обратить внимание, что приведенный шаблон сервлета содержит только аннотации обработки запросов типа GET и POST. Это сделано для того, чтобы его работу можно было проверить с помощью приложения браузера.

Для демонстрации того, что класс ***LetsRestService*** является RESTful-сервисом, создадим в каталоге ***WebContent*** нашего проекта HTML-файл с именем ***index.html***, как это показано на листинге 6.2.

Листинг 6.2 — Исходный файла index.html проекта lab9

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="UTF-8">
	<title>Тест lab9</title>
</head>
<body>
```

```
        <h2>Тестирование приложения типа JAX-RS</h2>
        <hr>
        <a href="http://localhost:8080/lab9/test">
        Проверка связи с сервисом: .../lab9/test</a>
        <hr>

        <a href="http://localhost:8080/lab9/letter">
              Получение списка записей: .../lab9/letter</a>
        <hr>

              Получение записи по идентификатору: .../lab9/letter/1
        <form name="f0" action="http://localhost:8080/lab9/letter/1"
                    method="get" accept-charset="UTF-8">

              <input type="text" size="6">
              <input type="submit">
        </form>
        <hr>

        Добавление новой записи: .../lab9/letter <br>
        <form name="f1" action="http://localhost:8080/lab9/letter"
                               method="post" accept-charset="UTF-8">

              <textarea rows="2" cols="40" name="text"></textarea>
              <input type="submit">
        </form>
        <hr>

        Удаление записи по идентификатору: .../lab9/delete
        <form name="f2" action="http://localhost:8080/lab9/delete"
                    method="post" accept-charset="UTF-8">

              <input type="text" size="6">
              <input type="submit">
        </form>
        <hr>

        Модификация записи по идентификатору: .../lab9/mod
        <form name="f3" action="http://localhost:8080/lab9/mod"
                    method="post" accept-charset="UTF-8">

              <input type="text" size="6">
              <textarea rows="2" cols="40" name="text"></textarea>
              <input type="submit">
        </form>
        <hr>
</body>
</html>
```

Обратите внимание, что получение списка записей и добавление новой записи осуществляются по одному адресу ***http://localhost:8080/lab9/letter***, но разными методами ***GET*** и ***POST***.

Теперь, если в проекте ***lab9*** выделить вкладку с файлом ***index.html*** и запустить сервер, то мы получим окно браузера для тестовых испытаний, что показано на рисунке 6.1.

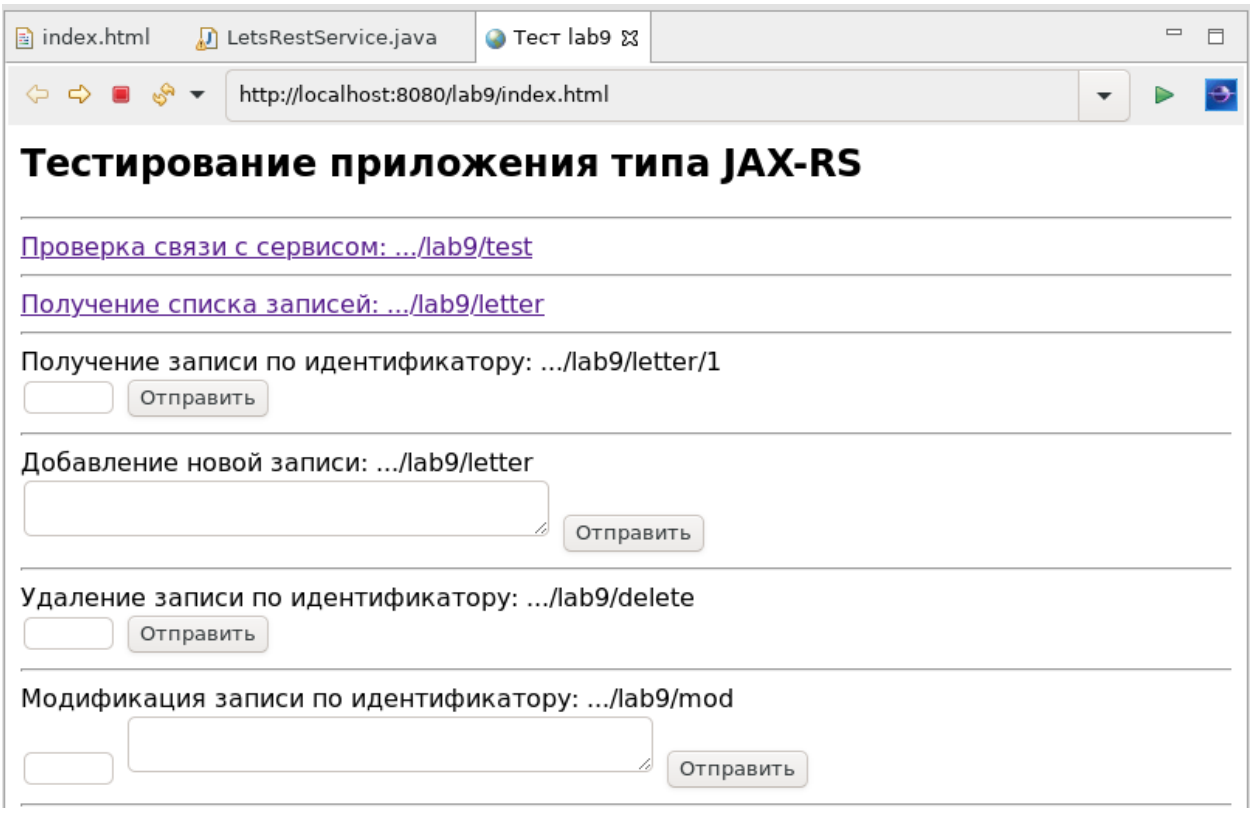

Рисунок 6.1 — Окно браузера для тестовых испытаний сервлета LetsRestService

Если активировать ссылку «***Проверка связи с сервером: .../lab9/test***», то получим результат, показанный на рисунке 6.2.

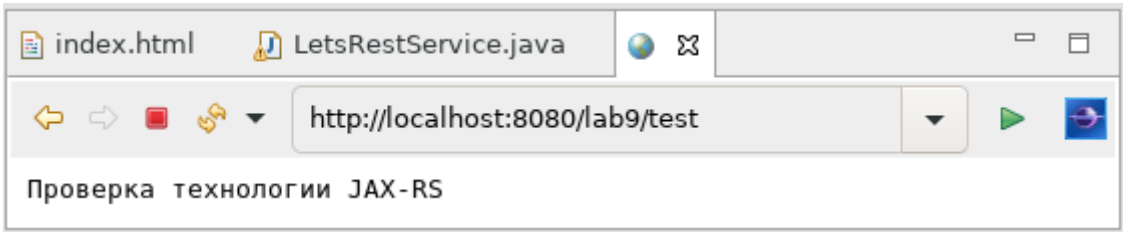

Рисунок 6.2 — Ответ сервлета LetsRestService

Другие ссылки файла ***index.html*** возвратят аналогичный результат, хотя и с другим содержанием, соответствующим назначению этих ссылок.

Таким образом, работоспособность сервиса RESTful — вполне показана представленными примерами.

Общий адекватный ответ RESTful-сервиса формируется с помощью специального класса ***javax.ws.rs.core.Response***.

В приведенном выше шаблоне класса ***LetsRestService***, формат ответа

представлен в виде обычного текста, что определяется и обеспечивается соответсвующей аннотацией ***@Produces(MediaType.TEXT_PLAIN)***. Для корректного формирования ответа сервера, необходимо использовать объект класса ***Response***, который учитывает все особенности функционирования протокола HTTP. В таблице 6.5 представлены основные методы этого класса, используемые для указанной цели.

Таблица 6.5 — Основные методы класса Response [17]

Метод	***Описание***
accepted()	Создает новый объект ResponseBuilder с состоянием: ***202 — Принято***.
created()	Создает новый объект ResponseBuilder для созданного ресурса, с использованием его URI.
noContent()	Создает новый объект ResponseBuilder для пустого ответа.
notModified()	Создает новый объект ResponseBuilder с состоянием: ***304 — Не изменялось***.
ok()	Создает новый объект ResponseBuilder с состоянием: ***200 — Хорошо***.
serverError()	Создает новый объект ResponseBuilder с состоянием: ***500 — Серверная ошибка***.
status()	Создает новый объект ResponseBuilder с предоставленным состоянием.
temporaryRedirect()	Создает новый объект ResponseBuilder с временным перенаправлением.
getCookies()	Получает cookie из сообщения ответа.
getHeaders()	Получает заголовки из сообщения ответа.
getLinks()	Получает ссылки, прикрепленные к сообщению в заголовке.
getStatus()	Получает код состояния, ассоциированный с ответом.
readEntity()	Получает объект сообщения, как экземпляр указанного типа Java, используя интерфейс MessageBodyReader, поддерживающий отображение сообщения на запрошенный тип.

Уже представленный перечень аннотаций показывает достаточно проработанный инструментарий проекта JAX-RS. К ним необходимо еще добавить:

а) список аннотаций параметров запроса: ***@DefaultValue***, ***@PathParam***, ***@QueryParam***, ***@FormParam***, ***@MatrixParam***, ***@HeaderParam*** и ***@CookieParam***;

б) ***@Context*** — аннотация, возвращающая весь контекст объекта запроса.

Учитывая достаточно большой объем имеющихся наработок, ограничимся только рядом примеров, рассмотренных в следующем подразделе.

6.2 Реализация Web-службы в стиле REST

Web-службы в стиле REST — это всего лишь один из вариантов сетевых программных оболочек, обслуживающих серверные приложения поставщиков сервисов.

Приведенный тезис призван подчеркнуть, что все технологические новинки сервис-ориентированных систем призваны обеспечить удобный и адекватный сервис к достаточно объемным прикладным системам.

В нашей дисциплине, учебной прикладной системой является хранилище записей, содержашее сущности класса ***Letter***. Сама система была представлена как сервисная EJB-компонента в различных видах: как классы ***Letters*** и ***Lets2*** (см. листинги 3.4 и 3.24 главы 3), ***ListLetters*** (см. листинг 4.4 главы 4) и ***Lets7*** (см. листинг 5.5 главы 5). В данной главе, указанная выше традиция продолжается применительно к технологии RESTful.

Не имея возможности подробно рассмотреть все аспекты технологических возможностей проекта JAX-RS, мы ограничимся демонстрацией следующих решений:

1) ***преобразуем*** сущность ***Letter*** к виду, достаточному для использования ее стиле REST;
2) ***создадим*** EJB-компоненту ***Lets9***, фактически скопировав ее с аналогичной компоненты ***Lets7***;
3) ***интегрируем*** EJB-компоненту ***Lets9*** в сервисный класс ***LetsRestService***, обеспечив его необходимым прикладным функционалом;
4) ***реализуем*** ряд сервисов из представленного шаблона ***LetsRestService***, ограничиваясь только методами GET и POST для вывода результатов работы сервисов в окно приложения-браузера.

6.2.1 Преобразование сущности Letter

Аннотации сущностей технологии JPA являются совместимыми с аннотациями XML-представлений технологии JAXB.

Создаем в проекте ***lab9*** класс с именем ***Letter*** и заменяем его содержимое на тело аналогичного класса из проекта ***lab4***.

Вносим в текст класса ***Letter*** добавления, соответствующие технологии JAXB, как это показано на листинге 6.3.

Обратите внимание, что аннотации ***@XmlElement*** ставятся перед публичными методами ***get*()***, а не перед приватными переменными.

Листинг 6.3 — Преобразованный класс Letter.java для проекта lab9

```
package rsos.lab9;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

/**
 * Базовая сущность Letter, использующая аннотации
 * @Entity и @Table(...).
 * Добавляются: обязательная аннотация @XmlRootElement
 * и @XmlType(...)
 */
@XmlType(propOrder={"id", "date", "name", "text"})
@XmlRootElement
@Entity
@Table(name = "t_letter")
public class Letter implements Serializable
{
	// Идентификатор сериализации
	private static final long serialVersionUID = 1L;

	@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;

	@Column(nullable = false)
	@Temporal(TemporalType.TIMESTAMP)
	private Date date;
	private String name;
	@Column(length = 4096)
	private String text;

	/**
	 * Конструкторы, геттеры, сеттеры и другие бизнес-методы
	 */
	public Letter() {}

	public Letter(Date date, String name, String text)
		{
	  this.date = date;
	  this.name = name;
	  this.text = text;
	}
	// Геттеры и сеттеры POJO-класса
	/**
	 * Аннотация @XmlElement для публичного метода
```

```
 * getId()
 */
@XmlElement
public int getId() {
	return id;
}

public void setId(int id) {
	this.id = id;
}
/**
 * Аннотация @XmlElement для публичного метода
 * getDate()
 */
@XmlElement
public Date getDate() {
	return this.date;
}

public void setDate(Date date) {
	this.date = date;
}
/**
 * Аннотация @XmlElement для публичного метода
 * getName()
 */
@XmlElement
public String getName() {
	return name;
}

public void setName(String name) {
	this.name = name;
}
/**
 * Аннотация @XmlElement для публичного метода
 * getText()
 */
@XmlElement
public String getText() {
	return text;
}

public void setText(String text) {
	this.text = text;
}
```

После проведенных изменений, сущность ***Letter*** становится способной не только обеспечивать ORM-отображение в таблицу ***t_letter*** базы данных ***lab4db***, но и преобразовываться в XML-представление, необходимое для передачи по сети. Следует также помнить, что теперь, перед запуском программ проекта ***lab9***, необходимо стартовать СУБД Apache Derby.

6.2.2 Реализация EJB-компоненты Lets9

EJB-компоненты программной платформы Java EE способны инкасулировать менеджеры сущностей и сами инкапсулироваться в сервлеты сервисов RESTful.

Принципиально, RESTful-сервисы могут быть реализованы на основе прямого использования классов сущностей, но следует заметить, что такое решение приемлемо только для очень простых примеров, иначе программное обеспечение сервлетов станет очень громоздким и запутанным. Следует также учесть, что сервис может использовать множество классов сущностей. Это зависит от объема и содержания задач, решаемых приложением.

Главная проблема реализации приложений уровня предприятий — это их сложность и разноплатовость. В таких условиях, любое проектное решение, направленное на упрощение логики и объема исходного кода приложений играют положительную роль, повышающую надежность результата и снижение затрат на сопровождение уже реализованных проектов.

Как уже было отмечено ранее, основную роль в упрощении проектных решений играют аннотации и функционал служебных сервисов, поддерживаемых контейнерами программной платформы Java EE. Сервис EJB-компонент, также играет здесь свою позитивную роль.

Стандартизация программного обеспечения EJB-компонент, применительно к стилю проектирования RESTful, связана с поддержкой этих компонент набором методов, соответствующих набору операций CRUD.

В предыдущей главе, для демонстрации Web-служб SOAP нами использовалась EJB-компонента ***Lets7***, успешно реализующая все необходимые операции CRUD применительно к сущности ***Letter***.

Создадим в проекте ***lab9*** EJB-компоненту с именем ***Lets9***, скопировав исходный текст класса ***Lets7*** и проведя необходимые минимальные изменения, как это показано на листинге 6.4.

Листинг 6.4 — EJB-компонента Lets9.java для проекта lab9

```
package rsos.lab9;
import java.util.List;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
```

```java
@Stateless
@LocalBean
public class Lets9
{
	// Доступ к контексту менеджера сущностей
	@PersistenceContext(name = "lab4-unit2")
	private EntityManager em;

	// Получение списка записей таблицы t_letter
	public List<Letter> getList()
	{
		// System.out.println("Lets9:getList()...");
		// Этап 1. Создаем объект builder
		CriteriaBuilder builder =
			em.getCriteriaBuilder();

				// Этап 2. Создаем объект criteria
		CriteriaQuery<Letter> criteria =
				builder.createQuery(Letter.class);

		// Этап 3. Создаем объект root
		Root<Letter> root =
			criteria.from(Letter.class);

		// Этап 4. Преобразовываем объект criteria,
		// включая использование объекта root
		criteria.select(root);

		// Этап 5. Добавляем сортировку
		criteria.orderBy(builder.desc(root.get("date")));

		//Этап 6. Формируем запрос в виде объекта query
		TypedQuery<Letter> query = em.createQuery(criteria);
		//Этап 7. Получаем результат
		List<Letter> list =
					query.getResultList();

		//Этап 8. Обрабатываем результат
		return list;
	}

	// Получение объекта по ключу
	public Letter getLetter(int id)
	{
		Letter l =
			em.find(Letter.class, new Integer(id));

		return l;
	}

	// Добавить объект в базу данных
	public void addLetter(Letter letter) {
		em.persist(letter);
	}

	// Удалить объект из базы данных по ключу
```

```
	public void deleteLetter(int id)
	{
		// Проверка наличия аргумента вызова
		Letter letter =
			em.find(Letter.class, new Integer(id));
			if(letter == null)
				return;

		// Непосредственное обращение на удаление объекта
		em.remove(letter);
	}

	// Модифицировать объект в базе данных
	public void modLetter(Letter letter) {
		if(letter == null)
					return;
		Integer Id = letter.getId();

		Letter l =
			em.find(Letter.class, Id);
		if(l == null)
				return;

		//System.out.println("Letters: модифицирую №" + Id);
		l.setDate(letter.getDate());
		l.setName(letter.getName());
		l.setText(letter.getText());
	}
}
```

Обратите внимание, что мы удалили ненужные для проекта интерфесы, оставив только аннотации:

а) ***@Stateless*** и ***@LocalBean***, определяющие EJB-компоненту;

б) ***@PersistenceContext(...)*** - для подключения менеджера сущностей технологии JPA.

EJB-компонента ***Lets9*** предоставляет сервис в виде списка объектов типа ***Letter***.

На листинге 6.3, сущность ***Letter*** уже преобразована для использования ее технологией JAXB. Теперь необходимо определить структуру объектов, представляющих список объектов типа ***Letter***. Для этой цели введем в проект ***lab9*** новый аннотированный класс с именем ***ListLets***, показанный на листинге 6.5.

Листинг 6.5 — Исходный текст класса ListLets.java проекта lab9

```
package rsos.lab9;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
```

```
import javax.xml.bind.annotation.XmlRootElement;

/**
 * Корневая аннотация для всего класса, задающая
 * имя списка объектов типа Letter.
 */
@XmlRootElement(name="letters")
public class ListLets implements Serializable
{
    private static final long serialVersionUID = 1L;

    // Приватная переменная класса
    private List<Letter> list = new ArrayList<>();
    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
    public ListLets() {}

    public ListLets(Letter letter) {
      if(letter != null)
            list.add(letter);
    }

    public ListLets(List<Letter> list) {
           this.list = list;
    }
    /**
     * Аннотация для публичного метода getList(), задающая имя для
     * описания объекта типа Letter и допускающая пустой список.
     */
    @XmlElement(name="letter", nillable = true)
    public List<Letter> getList() {
           return list;
    }
}
```

На этом, подготовительную часть проекта ***lab9*** можно считать законченой и EJB-компоненту ***Lets9***, а также подобные ей, необходимо инкапсулировать в RESTful-сервлет с помощью аннотации ***@EJB***.

6.2.3 Получение списка записей в формате XML

Реальный RESTfull-сервис может инкапсулировать множество различных EJB-компонент.

В нашем учебном примере используется только одна EJB-компонента ***Lets9***, но она должна быть протестирована перед реализацией любого прикладного сервиса. Проведем такое тестирование, инкапсулировав ***Lets9*** в сервлет ***LetsRestService***, и изменив метод ***testRS()***, как это показано на листинге 6.6.

Листинг 6.6 — Инъекция EJB-компоненты Lets9 в сервлет LetsRestService

```
/**
 * Аннотации, определяющие сервлет типа RESTful, который
 * доступен по адресу: http://localhost:8080/lab9
 */
@Path("/")
@Stateless
public class LetsRestService
{
	// Инъекция EJB-компоненты
	@EJB
	Lets9 lets;

	// Простой тест
	/**
	 * Простейшая демонстрация доступа по методу GET, доступная
	 * по адресу: http://localhost:8080/lab9/test
	 */
	@GET
	@Path("/test")
	@Produces(MediaType.TEXT_PLAIN)
	public String testRS()
	{
		if(lets == null)
			System.out.println("Объект типа Lets9 - не существует!");
		else
			System.out.println("Объект типа Lets9 - подключен!");

		return "Проверка технологии JAX-RS";
	}
```

Теперь, если в проекте ***lab9*** выделить в кладку ***index.html***, запустить сервер и активировать ссылку «***Проверка связи с сервером: .../lab9/test***», то в консоле среды разработки Eclipse EE получим результат, показанный на рисунке 6.3.

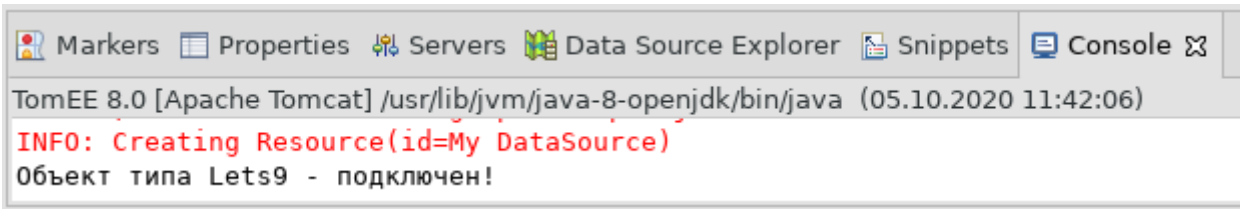

Рисунок 6.3 — Ответ сервлета LetsRestService

Ответ сервера показывает, что EJB-компонента ***Lest9*** успешно инкапсулируется в сервлет ***LetsRestService***.

Теперь, преобразуем метод ***getLets()*** сервлета ***LetsRestService***, согласно содержимому листинга 6.7.

Листинг 6.7 — Новый метод getLets() сервлета LetsRestService

```
// Получение списка записей
/**
 * Доступ по методу GET, читающий весь список записе, при
 * обращении по адресу: http://localhost:8080/lab9/letter
 */
@GET
@Path("/letter")
public Response getLets()
{
      return Response
            .ok(new ListLets(lets.getList()),
                  MediaType.APPLICATION_XML)
            .build();
}
```

Если теперь в проекте ***lab9*** можно выделить в кладку ***index.html***, запустить сервер и активировать ссылку «***Получение списка записей: .../lab9/letter***», то в окне браузера Eclipse EE получим результат, показанный на рисунке 6.4.

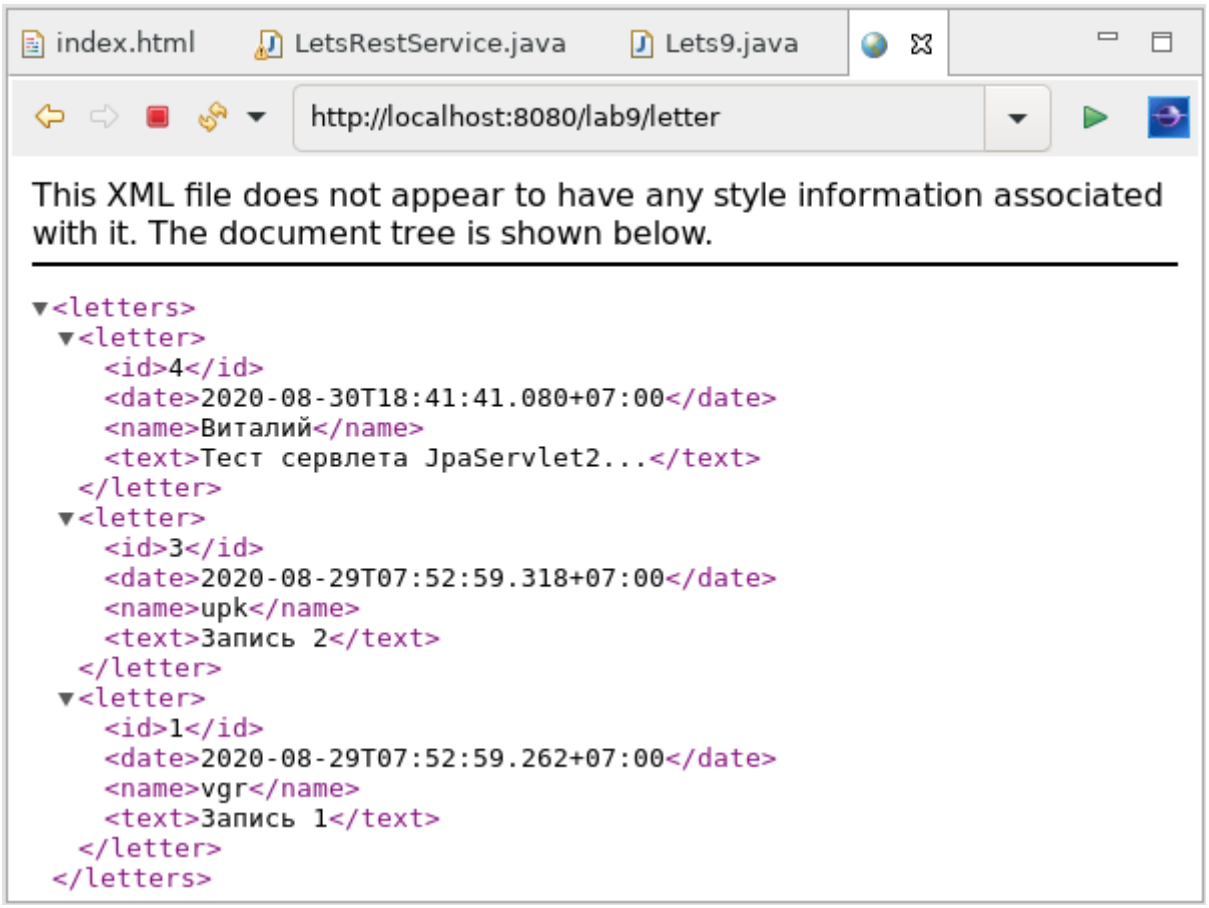

Рисунок 6.4 — Список записей объектов Letter в формате XML

6.2.4 Получение записи по номеру индентификатора

Стиль программирования REST рекомендует указывать доступ к отдельным объектам сервиса посредством прямой адресации URL.

Следуя требованиям стиля REST (см. пункт 6.1.2), рассмотрим чтение записи типа ***Letter***, например, по адресу: http://localhost:8080/lab9/letter/1.

Семантически, такой адрес указывает на запись таблицы ***t_letter*** с ключем ***id=1***, а, в пределах сервлета ***LetsRestService***, это будет интерпретироваться как адрес: http://localhost:8080/lab9/letter с параметром ***id***, который доступен с помощью аннотации ***@PathParam(...)***.

С учебной, да и с прикладной, точек зрения, отдельную запись удобнее представлять как элемент списка, потому что, если запись с заданным идентификатором не существует, то мы в окне браузера увидим пустой список, а не пустой экран.

С учетом перечисленных условий, для получения отдельной записи из таблицы ***t_letter***, преобразуем метод ***getLetter(...)*** сервлета ***LetsRestService***, согласно содержимому листинга 6.8.

Листинг 6.8 — Новый метод getLetter(...) сервлета LetsRestService

```
// Получение записи по идентификатору id=1
/**
 * Доступ по методу GET, читающий только одну запись, при
 * обращении по адресу: http://localhost:8080/lab9/letter/1
 * Должен использоваться один параметр типа int.
 */
@GET
@Path("/letter/{id : \\d+}")
public Response getLetter(@PathParam("id") int id)
{
      return Response
                  .ok(new ListLets(lets.getLetter(id)),
                              MediaType.APPLICATION_XML)
                  .build();
}
```

Обратите особое внимание, каким образом задается шаблон для целого числа в аннотации ***@Path()*** и как с помощью аннотации ***@PathParam()*** определяется тип аргумента в методе ***getLetter(...)***.

Если теперь в проекте ***lab9*** выделить вкладку ***index.html***, запустить сервер и активировать ссылку «***Получение новой записи по идентификатору: .../lab9/letter/1***», то в окне браузера Eclipse EE получим результат, показанный на рисунке 6.5, а если обратиться по адресу: http://localhost:8080/lab9/letter/2, то мы получим пустой список, как показано на рисунке 6.6.

Рисунок 6.5 — Запись объекта Letter с id=1 в формате XML

Рисунок 6.6 — Запись объекта Letter с id=2 в формате XML

6.2.5 Добавление новой записи

Стиль программирования REST не обеспечивает прямой адресацией URL всех потребностей приложений.

Рассмотрим пример добавления новой записи к списку объектов типа ***Letter***, который показывает ограниченность стиля прямой адресации объектов только с помощью одного URL.

Действительно, как уже сказано выше, для добавления новой записи следует использовать запрос типа ***POST***, поэтому, согласно стилю программирования REST, обращение к RESTful-сервису необходимо выполнять по адресу: http://localhost:8080/lab9/letter. А как в таком случае передавать текст сообщения? Поэтому, хотим мы или не хотим, но необходимо использовать дополнительные параметры. На листинге 6.9 показан вариант такого решения.

Листинг 6.9 — Новый метод addLetter(...) сервлета LetsRestService

```
// Добавление новой записи
/**
 * Демонстрация метода POST, добавляющего одну запись, при
 * обращении по адресу: http://localhost:8080/lab9/letter
 * Должен использоваться один параметр типа String.
 */
@POST
@Path("/letter")
@Produces(MediaType.TEXT_PLAIN)
public String addLetter(@FormParam("text") String text)
{
      // Для добавления новую записи, ее нужно создать
      Letter let =
                  new Letter(new Date(), "vgr", text);
      // Теперь, добавляем
      lets.addLetter(let);

       return "Добавлена новая запись\nПроверьте чтением списка записей!";
}
```

Обратите внимание, что поскольку запрос по методу POST выполняется из HTML-конструкции ***<FORM>***, то для чтения переданного параметра ***text*** используется аннотация ***@FormParam()***, указанная перед объявлением аргумента в методе ***addLetter(...)***. Сам текст ввода новой записи показан на рисунке 6.7.

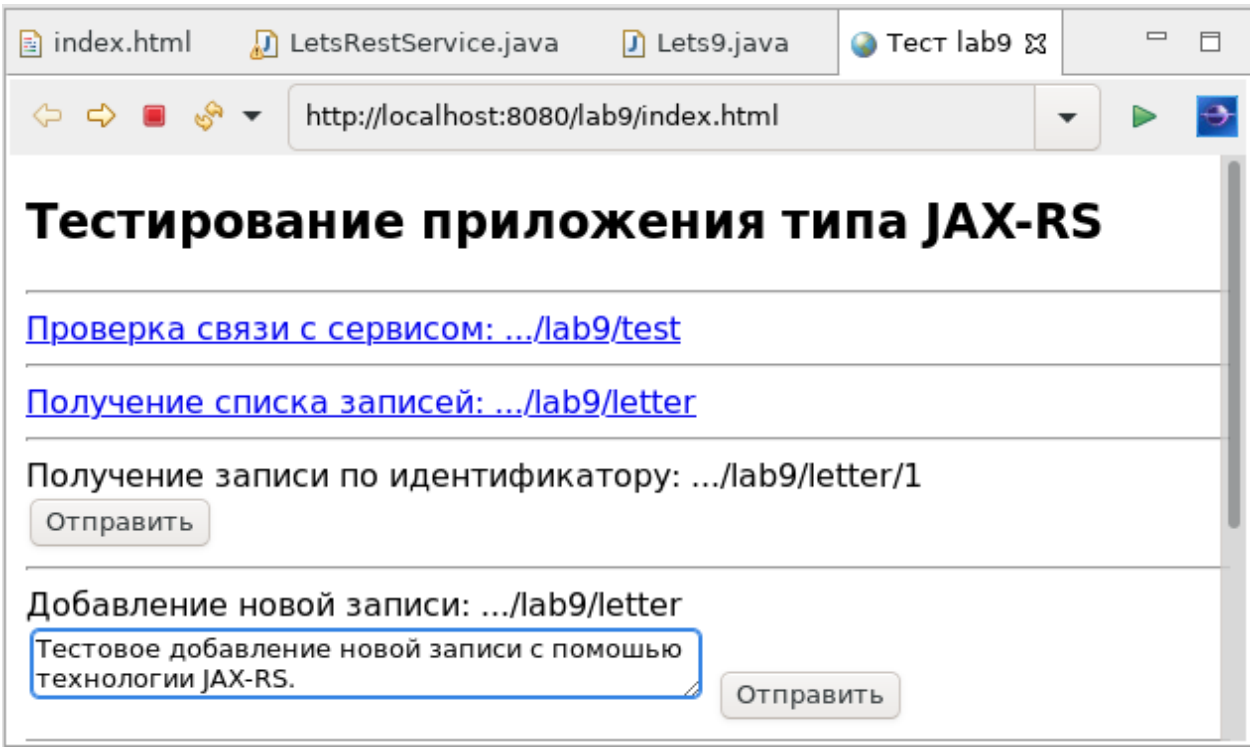

Рисунок 6.7 — Окно браузер, добавляющего новую запись

После активации кнопки «***Отправить***», новая запись будет создана и подтверждена текстовым сообщением, показанным на рисунке 6.8.

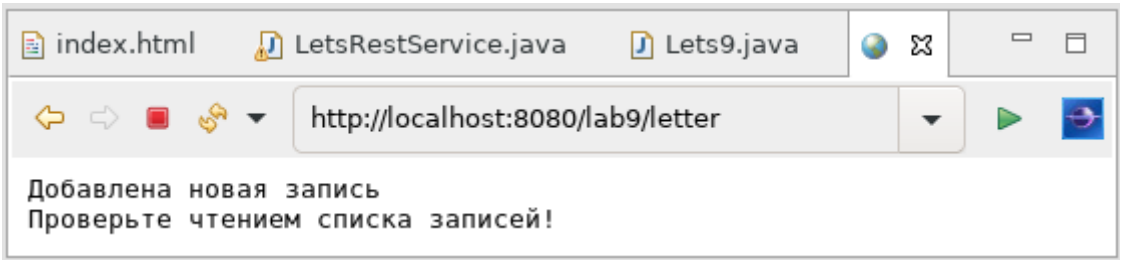

Рисунок 6.8 — Сообщение, подтверждающее добавление новой записи

Заново прочитав весь список сообщений, как это показано на рисунке 6.9, мы можем убедиться, что новая запись — добавлена.

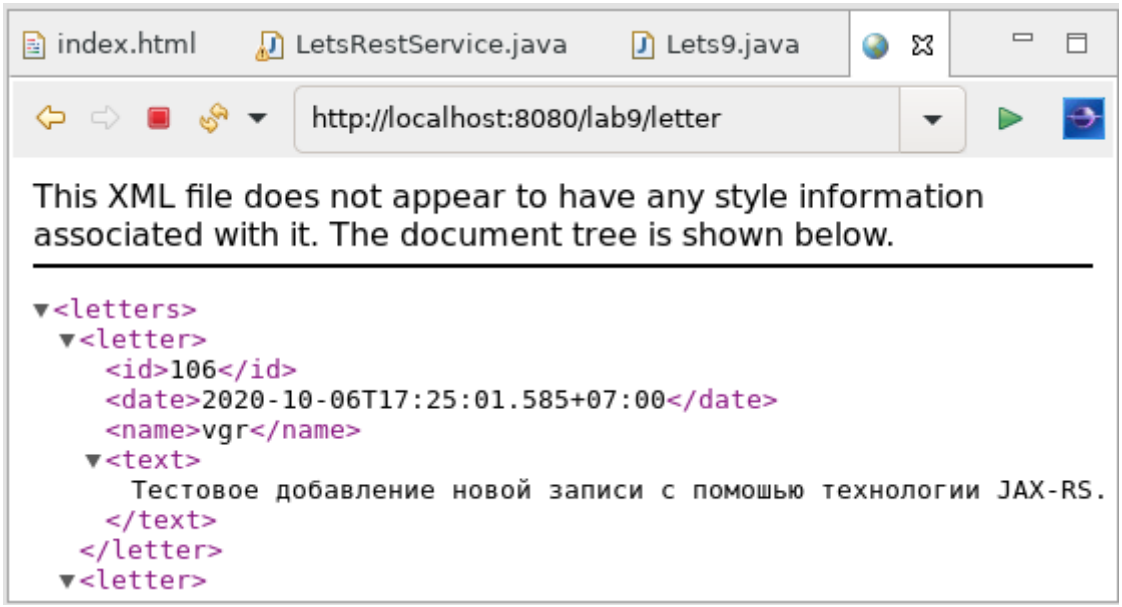

Рисунок 6.9 — Представление добавленой записи в формате XML

Этим примером мы завершаем тестирование RESTful-сервиса средствами клиентского агента в виде программы браузера. Хорошо видно, что возможности такого тестирования — весьма ограничены.

В следующем подразделе, рассмотрим инструментальные возможности, предоставляемые проектом JAX-RS для потребителей сервисов.

6.3 Вызов Web-служб в стиле REST

Наиболее мощные средства проекта JAX-RS сосредоточены в клиентской части потребителей сервисов.

В мае 2013 года появилась реализация JAX-RS версии 2.0, соответствующая спецификации JSR 339 и содержащая полноценный API для реализации клиентской части технологии RESTful. С этого момента отпала необходимость программирования запросов методами PUT и DELETE с помощью низкоуровневого API инструментальной платформы Java. Таким образом, были созданы необходимые инструментальные средства для реализации различного рода многозвенных архитектур на программной платформе Java EE, в которых сервера приложений могли эффективно обращаться к другим серверам RESTful-сервисов, предоставляя конечным клиентским приложениям, таким как браузеры, использовать только методы GET и POST.

Новый клиентский инструментарий проекта JAX-RS сосредоточен в пакете ***javax.ws.rs.client***.

Основные классы и интерфейсы пакета ***javax.ws.rs.client*** представлены в таблице 6.6.

Таблица 6.6 - Классы и интерфесы пакета javax.ws.rs.client [17]

Класс/интерфейс	***Описание***
Client	Класс основного объекта для API построения и выполнения клиентских запросов и анализа полученных ответов.
ClientBuilder	Фабрика для клиентского API, используемая для начального построения экземпляров объектов типа ***Client***.
Configurable	Клиентская конфигурационная форма для настройки объектов типа ***Client***, ***WebTarget*** и ***Invocation***.
Entity	Класс для получения содержимого ответов сервера.
Invocation	Итоговый запрос, готовый к исполнению.
Invocation.Builder	Построитель вызовов клиентского интерфейса.
WebTarget	Цель ресурса, идентифицируемая URI ресурса.

Используя объекты указанных в таблице типов, программные агенты клиентов формируют запросы, удовлетворяющие стилю RESTful, а затем проводят анализ полученных ответов.

Учебная цель данного подраздела — завершить демонстрацию инструментальных средств проекта JAX-RS, реализуя сетевое взаимодействие про-

граммного обеспечения как поставщиков, так и потребителей сервисов.

Для достижения указанной цели, учебный материал данного подраздела разделен на пять частей:

а) ***в первой и второй частях*** — дается описание инструментальных средств потребителей сервисов и приводится полная реализация класса сервлета ***LetsRestService***, ориентированного на взаимодействие с пользовательскими агентами;

б) ***в третьей части*** — на основе проекта ***labs*** реализуется шаблон пользовательского агента для взаимодействия с сервисом проекта ***lab9***;

в) ***последние две части*** — демонстрируют реализацию доступа к сервисам с использованием запросов типа GET, POST, PUT и DELETE.

6.3.1 Инструментальные средства потребителя сервиса

Основой запросов потребителей сервисов являются отдельные объекты типа ***Client***, которые создаются с использованием статических методов фабрики ***ClientBuilder***, например:

```
Client client = ClientBuilder.newClient();
```

Для того, чтобы указать цель запроса, создается целевой объект типа ***WebTarget***, например:

```
WebTarget target = client.target("http://localhost:8080/lab9/letter");
```

К объекту типа ***WebTarget*** применимы ряд методов, уточняющие адрес запроса или формирующие параметры запроса, например:

```
target.path("1");
target.queryParam("name", "Содержимое параметра");
```

Закончив формирование нужной цели, следует создать объект типа ***Invocation.Builder***, причем можно указать и ***MediaType*** запроса, например:

```
Invocation.Builder builder =
        target.request();
Invocation.Builder builder =
        target.request(MediaType.TEXT_PLAIN);
```

Завершается формирование запроса созданием одного из вариантов

объекта типа ***Invocation***, в котором должен быть указан один из доступных типов запроса, например:

```
Invocation invocation =
        builder.buildGet();
Invocation invocation =
        builder.buildDelete();
Invocation invocation =
        builder.buildPost(Entity.entity(new Letter(...)));
Invocation invocation =
        builder.buildPut(Entity.entity(new Letter(...)));
```

Обратите внимание, что построение запросов для методов POST и PUT предполагает передачу поставщику сервиса объектов, оформленных как объекты типа ***Entity***.

Окончательно, сам запрос осуществляется методом ***invoke()***, предполагая получение ответа типа ***Response***, например:

```
Response response =
        invocation.invoke();
```

Отправив запрос и получив ответ в виде объекта типа ***Response***, потребитель сервиса может осуществить его предварительный анализ, например:

1) response.***getStatusInfo***() — получить код завершения запроса;
2) response.***getLength***() — определить полученную длину ответа;
3) response.***getDate***() — узнать дату формирования ответа;
4) response.***getHeaderString***("Content-type") — прочитать различные параметры заголовков ответа.

Главным в объекте типа ***Response*** является конечно тело сообщения, которое можно обрабатывать в виде объекта типа ***String*** или, если полностью известен тип объекта, то востановить его из текстового сообщения. Для этого используется метод ***readEntity(...)***. Например, если мы хотим анализировать строку, то ее можно извлечь следующим образом:

```
String body =
    response.readEntity(String.class);
```

Если нам известно, что полученный объект имеет тип ***Letter*** и у нас имеется описание этого класса, то следует записать:

Letter body =
response.***readEntity***(Letter.class);

Для примера, рассмотрим программу клиента, которая демонстрирует использование перечисленных выше инструментальных средств.

В качественном виде, такая программа обращается к поставщику сервиса, реализованного в проекте ***lab9*** и запрашивает у него полный список записей объектов типа класса ***Letter***, а затем — распечатывает полученный результат формата XML в виде строки.

Формальная постановка задачи — необходимо:

1) обратиться к поставщику сервиса http://localhost:8080/lab9/letter;
2) получить ответ в виде объекта типа ***Response***;
3) извлечь и распечатать текстовое содержимое ответа.

Реализация поставленной задачи включает:

1) создание нового проекта ***jaxrs*** типа ***Dynamic Web Project***;
2) создание нового класса с именем ***RunGetLetter***, содержимое которого показано на листинге 6.10.

Листинг 6.10 — Исходный текст класса RunGetLetter.java проекта jaxrs

```
package rsos.lab9;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class RunGetLetter
{
	public static void main(String[] args)
	{
		// Создание объекта типа Client
		Client client =
				ClientBuilder.newClient();

		// Создание цели запроса
		WebTarget target =
			client.target("http://localhost:8080/lab9/letter");
		//target.path("1");
		//target.queryParam("name", "Содержимое параметра");

		// Раздельное создание объектов builder и invocation
		Invocation.Builder builder =
				target.request(MediaType.TEXT_PLAIN_TYPE);

		Invocation invocation =
				builder.buildGet();

		// Отправка запроса и получение ответа
```

```
            Response response =
                        invocation.invoke();

            // Извлечение ответа в виде объекта строки
            String body =
                        response.readEntity(String.class);

            // Печать содержимого тела ответа на терминал
            System.out.println(body);

            // Закрытие объемных объектов
            response.close();
            client.close();
        }
}
```

Результат работы класса ***RunGetLetter*** показан на рисунке 6.10. Он конечно не производит впечатления, поскольку список объектов типа ***Letter*** представлен в формате XML и выводится в виде строки. Тем не менее, можно сравнить этот вывод с содержимым, например, рисунка 6.4, полученного при работе проекта ***lab9***, и убедиться в правильности полученного результата.

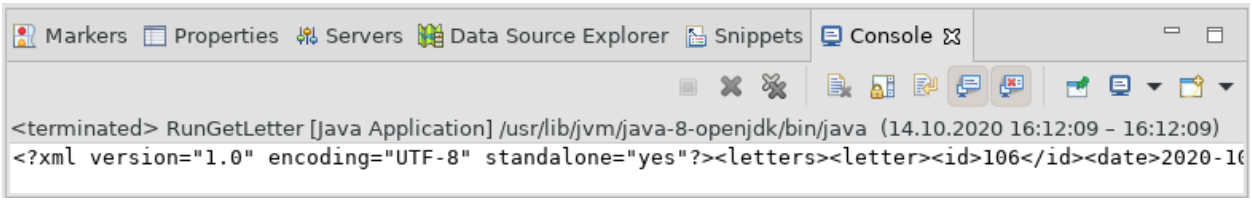

Рисунок 6.10 — Результат работы класса RunGetLetter

> В целом, Web-службы в стиле REST не предназначены для прямого вывода информации в приложения типа браузеров.

Подобное утверждение может показаться спорным, но, как видно из описанных инструментальных средств потребителей сервисов, полученные объекты типа ***Response*** требуют дополнительной обработки, если на стороне клиента имеется их описание.

Безусловно, в рамках программной платформы Java EE, RESTful-службы должны обеспечивать другие технологии представления информации, такие как ***JSF*** (*JavaServer Faces*). Именно компоненты-подложки технологии JSF должны выступать потребителями сервисов RESTful.

С другой стороны, Web-службы в стиле REST, зародившись как альтернатива классической технологии Web-служб SOAP, считаются более экономными и эффективными в плане взаимодействия между поставщиками и потребителями сервисов. В частности, это связано с сокращенным набором рекомендуемых сервисов, обозначаемых как операции типа CRUD.

Теперь, проведем демонстративную реализацию нашего учебного сервиса, используя указанные классические ограничения.

6.3.2 Полная реализация RESTfull-сервиса

Реализация RESTful-сервиса должна следовать ряду общих правил, обеспечивающих выполнение операций типа CRUD.

Учитывая большое многообразие возможных реализаций Web-служб в стиле RESTful, ограничимся рассмотрением эталонного примера, который поддерживает только набор операций типа CRUD. Для изучения других примеров следует воспользоваться специальной литературой или примерами удачных проектов, опубликованных в различных источниках.

Реализацию учебного примера проведем посредством модификации класса ***LetsRestService***, который был представлен в подразделе 6.1 и уже не раз подвергался модификации в подразделе 6.2.

Реализация операций типа CRUD на основе протокола HTTP должна удовлетворять требованиям спецификаций этого протокола.

Спецификации протокола HTTP определяют какие коды состояния требуется возвращать сервером потребителю сервиса при успешном и других вариантах ответов. В нашем случае, это касается методов GET, POST, PUT и DELETE.

Методы GET возвращают клиенту любую информацию в виде объекта или списка объектов, на которую указывает запрошенный URI. Во всех этих случаях метод GET должен возвращать код: ***200 — Хорошо***.

Метод POST используется для создания нового ресурса, идентифицируемого URI-запроса. Сервер должен принимать в качестве ресурса сохраняемый объект, а в ответ должны возвращаться коды:

1) ***код 201 — Создано с URI нового ресурса***, если метод создал ресурс, а в качестве тела должен возвратиться URI созданного ресурса;
2) ***код 204 — Нет содержимого***, если метод не создал ресурса, который можно было бы идентифицировать по URI.

Метод DELETE требует, чтобы сервер удалил ресурс, на который указывает содержащийся в запросе URI. Сервер должен возвращать один из трех вариантов ответа:

1) ***код 200 — Хорошо***, если в ответе содержится объект;
2) ***код 202 — Принято***, если действие пока не запущено;
3) ***код 204 — Нет содержимого***, если действие было запущено, но в ответе отсутствует объект.

Метод PUT ссылается на уже существующий ресурс, который необходимо обновить. Если обновляется существующий ресурс, то должен быть возвращен один из следующих кодов состояния:

1) ***код 200 — Хорошо***, если в ответе возвращается модифицированный объект;
2) ***код 204 — Нет содержимого***, если в ответе модифицированный объект отсутствует.

> Коды состояния протокола HTTP не характеризуют полностью прикладной результат выполненного запроса.

Все приведенные выше коды состояния считаются успешным выполнением запроса клиента. Обратите внимание, что в реальных сложных приложениях сам сервис передает конечную реализацию другим объектам, например, EJB-компонентам, которые реализуют запрос на более низких уровнях. Кроме того, запросы клиента сначала обрабатываются контейнерами Web-сервисов и могут сами генерировать ответы без участия RESTful-компоненты сервиса.

Чтобы устранить все возможные неопределенности, при реализации учебного сервиса ***LetsRestService***, будем предполагать, что:

1) ***все запросы*** и ***ответы*** сервиса реализуются в виде представления XML;
2) ***положительный*** прикладной ответ должен содержать тело объекта;
3) ***негативный*** прикладной ответ не содержит тело объекта.

Более точно алгоритм класса ***LetsRestService***, показан на листинге 6.11.

Листинг 6.11 — Полная реализация класса LetsRestService.java проекта lab9

```
package rsos.lab9;
import java.net.URI;
import java.util.Date;
import java.util.List;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriBuilder;
import javax.ws.rs.core.UriInfo;
```

```java
/**
 * Аннотации, определяющие сервлет типа RESTful, который
 * доступен по адресу: http://localhost:8080/lab9.
 * Формат представления данных - XML.
 */
@Path("/")
@Stateless
public class LetsRestService
{
	// Инъекция EJB-компоненты
	@EJB
	Lets9 lets;

	// Инъекция объекта контекста
	@Context
	UriInfo uriInfo;

	// Простой тест
	/**
	 * Простейшая демонстрация доступа по методу GET, доступная
	 * по адресу: http://localhost:8080/lab9/test
	 */
	@GET
	@Path("/test")
	@Produces(MediaType.TEXT_PLAIN)
	public String testRS()
	{
		if(lets == null)
			System.out.println("Объект типа Lets9 - не существует!");
		else
			System.out.println("Объект типа Lets9 - подключен!");

		return "Проверка технологии JAX-RS";
	}
	// Получение списка записей
	/**
	 * Доступ по методу GET, читающий весь список записе, при
	 * обращении по адресу: http://localhost:8080/lab9/letter.
	 * Всегда возвращает код 200 - Хорошо.
	 */
	@GET
	@Path("/letter")
	public Response getLets()
	{
		return Response
				.ok(new ListLets(lets.getList()),
					MediaType.APPLICATION_XML)
				.build();
	}
	// Получение записи по идентификатору id
	/**
	 * Доступ по методу GET, читающий только одну запись, при
	 * обращении по адресу: http://localhost:8080/lab9/letter/{id}
	 * Должен использоваться один параметр типа int.
	 * Возвращает код 200, если объект найден, иначе возвращает
	 * код 204 - Нет содержимого.
	 */
```

```
@GET
@Path("/letter/{id : \\d+}")
public Response getLetter(@PathParam("id") int id)
{
      Letter l =
                  lets.getLetter(id);
      if(l == null)
            return Response         // код 204
                        .noContent().build();

      return Response               // код 200
                  .ok(l, MediaType.APPLICATION_XML)
                  .build();
}
// Добавление новой записи
/**
 * Демонстрация метода POST, добавляющего одну запись, при
 * обращении по адресу: http://localhost:8080/lab9/letter
 * Должен получать объект типа Letter и сохранять его данные:
 * date и text.
 * После сохранения объекта в базе данных, необходимо прочитать
 * весь список объектов.
 * Первый объект в списке должен быть новым созданным, тогда
 * возвращаем код 201 - Создано с URI Нового ресурса и сам адрес
 * ресурса, иначе - только код 204 - Нет содержимого.
 */
@POST
@Path("/letter")
@Consumes(MediaType.APPLICATION_XML)
public Response addLetter(Letter let)
{
      // Сохраняем данные запроса
      Date   date = let.getDate();
      String text = let.getText();

      lets.addLetter(let);          // Сохраняем новый объект
      List<Letter> ls =
                  lets.getList();   // Поучаем список всех объектов
      if(ls.isEmpty())
            return Response         // код 204
                        .noContent().build();

      Letter l =
                  ls.get(0);        // Получаем первый элемент списка
      if(date != l.getDate() || !text.equals(l.getText()))
            return Response         // код 204
                        .noContent().build();

      // Создаем объект адреса нового ресурса
      String tt =
                  "/letter/" + new Integer(l.getId()).toString();
      URI uri = UriBuilder
                  .fromResource(LetsRestService.class)
                  .path(tt).build();
      return      // код 201 - Создан с URI нового ресурса
                  Response.created(uri).build();
}
```

```
// Удаление записи по идентификатору id
/**
 * Доступ по методу DELETE, удаляющий конкретную запись, при
 * обращении по адресу: http://localhost:8080/lab9/letter/{id}
 * Сначала запись читается и, если она отсутствует, то возвращается
 * код 204 - Нет содержимого.
 * Полученная запись сохраняется и удаляется из базы данных.
 * В результате, возвращается запись и код 200 - Хорошо.
 */
@DELETE
@Path("/letter/{id : \\d+}")
@Produces(MediaType.APPLICATION_XML)
public Response deleteLetter(@PathParam("id") int id)
{
      Letter let =
                  lets.getLetter(id);
      if(let == null)
            return Response         // код 204
                        .noContent().build();

      // Удаляем запись
      lets.deleteLetter(id);

      return Response               // код 200
                  .ok(let, MediaType.APPLICATION_XML)
                  .build();
}
// Модификация записи по содержимому объекта
/**
 * Демонстрация метода PUT, изменяющего одну запись, при
 * обращении по адресу: http://localhost:8080/lab9/letter
 * Должен получать объект типа Letter.
 * После изменения объекта методом lets.modLetter(...), необходимо
 * прочитать измененный объект по его идентификатору id.
 * Если объект прочитан, то возвращаем его с кодом 200 - Хорошо,
 * иначе возвращаем код 204 - Нет содержимого.
 */
@PUT
@Path("/letter")
@Consumes(MediaType.APPLICATION_XML)
public Response modLetter(Letter let)
{
      if(let == null)               // Проверяем аргумент
            return Response         // код 204
                        .noContent().build();

      // Модифицируем запись
      lets.modLetter(let);

      Letter lx =
                  lets.getLetter(let.getId());
      if(lx == null)
            return Response         // код 204
                        .noContent().build();

      return Response               // код 200
                  .ok(lx, MediaType.APPLICATION_XML)
```

```
                    .build();
        }
}
```

Этим кодом мы завершаем разработку проекта ***lab9***, соотвествующего Web-службе поставщика сервиса.

Дальнейшие пункты данной главы посвящены приложению, реализующему функционал потребителя RESTful-сервиса.

6.3.3 Шаблон реализации потребителя сервиса

> Полная реализация RESTful-сервиса требует размещение функционала потребителя сервиса на сервере приложений.

Поскольку браузеры ориентированы только на HTTP-запросы GET и POST, нам необходимо серверное приложение, которое бы обеспечивало полную функциональность потребителя RESTful-сервиса.

В качестве основы такого приложения можно взять проект ***labs***, рассмотренный во второй главе и реализующий прототип учебного приложения по всем лабораторным работам данной дисциплины (см. подраздел 2.4). Хотя полная реализация этого проекта была отложена по причине очевидного усложнения учебных примеров, реализуемых в главах 3 - 5, тем не менее, для демонстрации агента потребителя сервиса Web-службы в стиле REST этот проект вполне подходит.

Напомню, что проект ***labs*** обеспечивает доступ к серверу приложений посредством адресации XHTML-файла соответствующей лабораторной работы (см. последний вариант реализации, описанный в пункте 2.4.8 и показанный на рисунке 2.32). В частности, работе ***№9*** соответствует файл ***lab9.xhtml***. Если за начальное представление этого файла взять содержимое листинга 6.12, то, после запуска проекта и авторизации пользователем ***upk***, внешний вид окна браузера будет отображен рисунком 6.11.

Листинг 6.12 — Исходное содержимое файла lab9.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

      <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">

      <!-- Переопределение контекстной страницы context -->
```

```
        <ui:define name="context">

        <div align="left" style="color:black;padding:10px">
              <b>lab9.xhtml - работа с сервисом:
              http://localhost:8080/lab9/letter</b>
              <hr/>
        </div>

        </ui:define>

        </ui:composition>

</html>
```

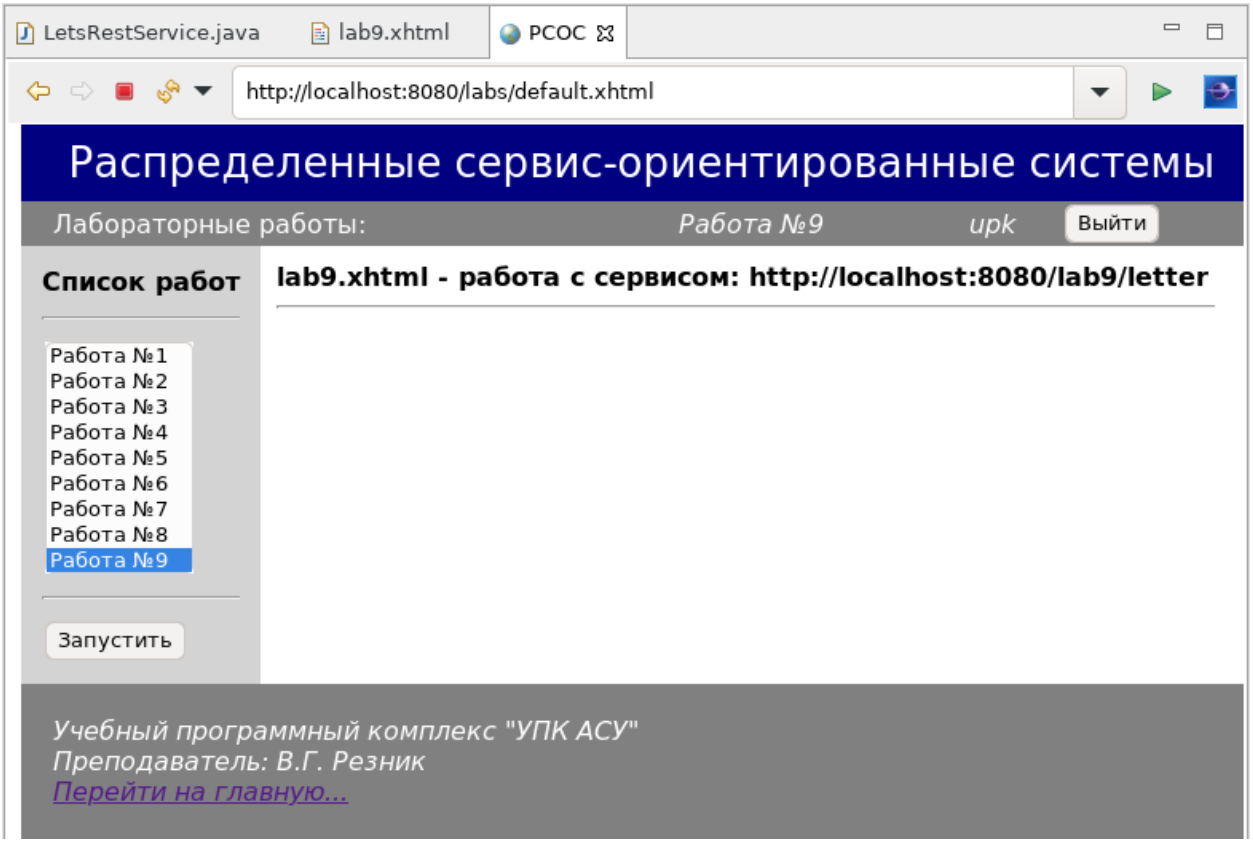

Рисунок 6.11 — Начальное отображение файла lab9.xhtml

Обратите внимание, что XHTML-ресурс ***lab9.xhtml*** не имеет активных элементов и еще не использует никакую компонету-подложку. Другими словами — это всего лишь «*заглушка*» для будущей реализации соответствующего проекта.

Вполне разумно для реализации шаблона потребителя сервиса использовать компонент-подложку с именем ***Lab9.java***, что создаст нужную семантическую ассоциацию при описании проекта.

Следующим шагом необходимо решить: какой функционал и область действия должна обеспечивать эта подложка ***Lab9.java***.

Чтобы правильно ответить на данный вопрос, студенту рекомендуется перечитать подраздел 2.4 второй главы. Особое внимание здесь необходимо обратить на пункт 2.4, где на рисунке 2.26 (см. стр. 114) представлена схема взаимодействия браузера и компонент-подложек JSF проекта ***labs***.

Для болшей наглядности изложения учебного материала, повторим отображение этого рисунка здесь, в виде рисунка 6.12.

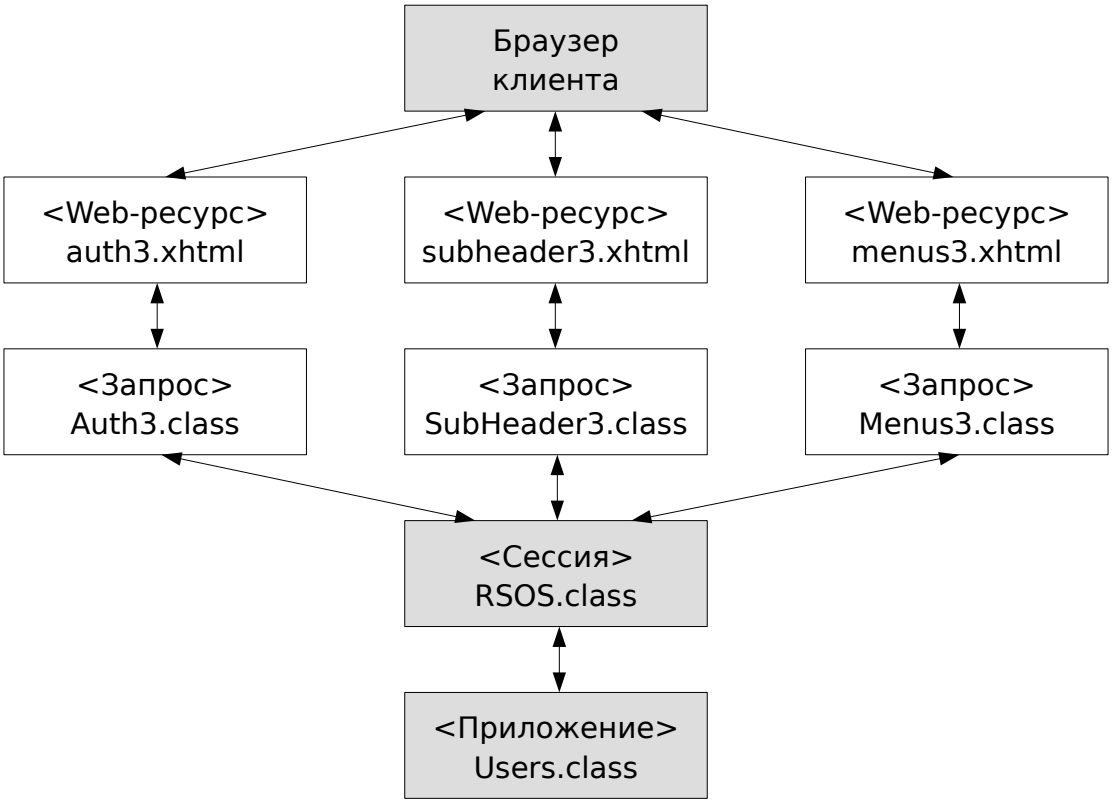

Рисунок 6.12 — Повторное отображение рисунка 2.26 главы 2, стр. 114

Хорошо видно, что все представленные XHTML-ресурсы имеют компоненты-подложки с областью действия ***@RequestScoped***. Это связано с тем, что они обслуживают локальные запросы к данным, которые храняться в CDI-компоненте ***RSOS.class*** с областью действия ***@SessionScoped***.

Особенность создаваемой здесь компоненты-подложки ***Lab9.class*** состоит в необходимости доступа к удаленным данным, реализуемым поставщиком сервиса в виде проекта ***lab9***, и сохранении этих данных для отображения их с помощью XHTML-ресурса ***lab9.xhtml***.

Из сказанного следует, что компонента ***Lab9.class*** должна обслуживать отдельного пользователя в течение всей сессии и иметь соответствующую аннотацию. Более того, эта компонента должна сохранять следующие данные:

а) **resMsg** — текстовое сообщение, соответствующее полученному объекту типа ***Response***;

б) **list** — список прочитанных сообщений типа ***List<Letter>***;

в) **id** — целочисленный идентификатор запроса объекта типа ***Letter***;

г) **text** — текстовое сообщение запроса для объекта типа ***Letter***;

д) **rsos** — инъекция сессионного объекта типа RSOS для получения данных о пользователе, выполняемой работе и результате авторизации;

е) **address** — текстовая константа http://localhost:8080/lab9/letter, соответствующая базовому адресу удаленного ресурса, который представлен сервером проекта ***lab9***.

Естественно, что компонента ***Lab9.class*** должна иметь соответствующий набор методов (***геттеров*** и ***сеттеров***), обслуживающих доступ к данным из XHTML-ресурса ***lab9.xhtml***.

Клиентская часть потребителя сервиса должна иметь описания классов ***Letter*** и ***ListLets***.

Действительно, поставщик сервиса, представленный Web-сервисом класса ***LetsRestService*** (см. листинг 6.11, стр. 272-276), возвращает объекты типа ***Letter*** и ***ListLets*** в формате JSON. Соответственно, клиентская сторона в виде класса ***Lab9*** должна восстанавливать эти объекты. Поэтому описания этих классов должны присутствовать в проекте ***labs***, но в клиентском исполнении, как это показано на листингах 6.13 - 6.14.

Листинг 6.13 — Клиентское описание класса Letter.java проекта labs

```
package asu.rsos;
import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

/**
 * Клиентская часть базовой сущности Letter.
 * Добавляются: обязательная аннотация @XmlRootElement
 * и @XmlType(...)
 */
@XmlType(propOrder={"id", "date", "name", "text"})
@XmlRootElement
public class Letter implements Serializable
{
```

```
// Идентификатор сериализации
private static final long serialVersionUID = 1L;

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private Date date;
private String name;
private String text;

/**
 * Конструкторы, геттеры, сеттеры и другие бизнес-методы
 */
public Letter() {
}

public Letter(Date date, String name, String text) {
    this.date = date;
    this.name = name;
    this.text = text;
}

// Геттеры и сеттеры POJO-класса
/**
 * Аннотация @XmlElement для публичного метода
 * getId()
 */
@XmlElement
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}
/**
 * Аннотация @XmlElement для публичного метода
 * getDate()
 */
@XmlElement
public Date getDate() {
    return this.date;
}

public void setDate(Date date) {
    this.date = date;
}
/**
 * Аннотация @XmlElement для публичного метода
 * getName()
 */
@XmlElement
public String getName() {
    return name;
}

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

```
    }
    /**
     * Аннотация @XmlElement для публичного метода
     * getText()
     */
    @XmlElement
    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    // Дополнительные функции форматирования
    public String toString() {
      return Integer.toString(id) + " " + date.toString() + " "
                  + name + " " + text;
    }

    public String getDateString() {
      SimpleDateFormat sdf =
                  new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      return sdf.format(this.date);
    }
}
```

Листинг 6.14 — Клиентское описание класса ListLets.java проекта labs

```
package asu.rsos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

/**
 * Корневая аннотация для всего класса, задающая
 * имя списка объектов типа Letter.
 */
@XmlRootElement(name="letters")
public class ListLets implements Serializable
{
    private static final long serialVersionUID = 1L;

    // Приватная переменная класса
    private List<Letter> list = new ArrayList<>();
    /**
     * Конструкторы, геттеры, сеттеры и другие бизнес-методы
     */
    public ListLets() {}

    public ListLets(Letter letter) {
      if(letter != null)
            list.add(letter);
    }
```

```
    public ListLets(List<Letter> list) {
        this.list = list;
    }
    /**
     * Аннотация для публичного метода getList(), задающая имя для
     * описания объекта типа Letter и допускающая пустой список.
     */
    @XmlElement(name="letter", nillable = true)
    public List<Letter> getList() {
        return list;
    }
}
```

Учитывая изложенное выше, главную часть схемы взаимодействия браузера, локальных агентов потребителя сервиса и удаленного сервера приложений поставщика сервиса можно представить рисунком 6.13.

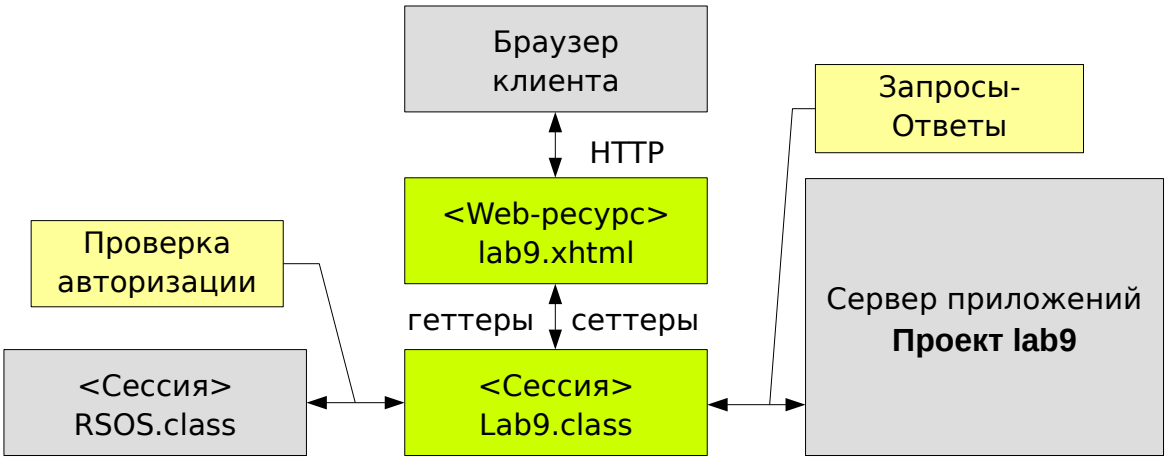

Рисунок 6.13 — Схема взаимодействия потребителя с поставщиком сервиса

Как видно из рисунка, сессионная компонента ***Lab9.class*** должна обеспечивать функционал трех видов:

1) ***isAuth()*** — метод запрашивающий результат авторизации пользователя и выбор работы - «*Работа №9*»;
2) ***геттеры/сеттеры*** — стандартные методы, обслуживающие запросы Web-ресурса ***lab9.xhtml***;
3) ***mGetList()***, ***mGetLetter()***, ***mPostLetter()***, ***mDeleteLetter()*** и ***mPutLetter()*** — методы, осуществояющие распросы к удаленному серверу приложений и возвращающие строку «***lab9***», что означает переход к Web-ресурсу ***lab9.xhtml***.

С учетом введенных ограничений, шаблон реализуемой компоненты ***Lab9.class*** представлен на листинге 6.15.

Листинг 6.15 — Шаблон файла Lab9.java проекта labs

```
package asu.rsos;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.enterprise.context.SessionScoped;
import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;

/**
 * Компонент-подложка ресурса lab9.xhtml, обеспечивающая
 * CRUD-операции потребителя сервиса, посредством запросов
 * GET, POST, DELETE и PUT к Web-сервису проекта lab9.
 */
@Named
@SessionScoped
public
class Lab9  implements Serializable
{
      /**
       * Стандартный идентификатор для сериализации
       */
      private static final long serialVersionUID = 1L;
      /**
       * Инъекция объекта класса RSOS
       */
      @Inject
      private RSOS rsos;
      /**
       * Приватные переменные, обслуживаемые методами
       * геттеров и сеттеров.
       */
      // Текстовое сообщение, соответствующее ответу типа Response
      private String resMsg = "Вводите запросы...";
      // Прочитанный список сообщений
      private List<Letter> list =
                  new ArrayList<>();
      // Идентификатор обрабатываемого сообщения
      private int id;
      // Текст обрабатываемого сообщения
      private String text;
      // Адрес доступа к сервису ListRestService проекта lab9
      private String address = "http://localhost:8080/lab9/letter";
      /**
       * Пустой конструктор
       */
      public Lab9() {}
      /**
       * Публичные методы геттеров и сеттеров
       */
      public String getResMsg() {return resMsg;}
      public void   setResMsg(String resMsg) {this.resMsg = resMsg;}

      public List<Letter> getList() {return list;}
```

```
//    public void   setList(List<Letter> list) {this.list = list;}

      public int  getId() {return id;}
      public void setId(int id) {this.id = id;}

      public String getText() {return text;}
      public void   setText(String text) {this.text = text;}

      public String getAddress() {return address;}
      public void   setAddress(String address) {this.address = address;}

      /**
       * Метод проверки авторизации
       */
      public boolean isAuth()
      {
            if(rsos.getUser() == null || rsos.getWork() == null)
                  return false;
            if(rsos.getUser().length() > 0
                  || "Работа №9".equals(rsos.getWork()))
                  return true;
            return false;
      }
      /**
       * Шаблоны методов, реализующих запросы к поставщику сервисов.
       * Все они должны возвращать строку "lab9", что возвращает
       * запрос к ресурсу lab9.xhtml.
       */
      //Получить список объектов типа Letter
      public String mGetList() {
            resMsg = "Метод GET - не реализован";
            return "lab9";
      }

      // Получить объект типа Letter по идентификатору id
      public String mGetLetter() {
            resMsg = "Метод GET - не реализован";
            return "lab9";
      }
      // Создать новый объект типа Letter
      public String mPostLetter() {
            resMsg = "Метод POST - не реализован";
            return "lab9";
      }

      // Удалить объект типа Letter по идентификатору id
      public String mDeleteLetter() {
            resMsg = "Метод DELETE - не реализован";
            return "lab9";
      }

      // Модифицировать объект типа Letter по идентификатору id
      public String mPutLetter() {
            resMsg = "Метод PUT - не реализован";
            return "lab9";
      }
}
```

Обратите внимание, что на приведенном листинге шаблона компоненты-подложки ***Lab9*** нереализованными являются только пять последних методов, отвечающих за запросы к Web-сервису проекта ***lab9***. Ими мы займемся в следующих пунктах данного подраздела, а сейчас преобразуем ресурс ***lab9.xhtml*** к виду, показанному на листинге 6.16.

Листинг 6.16 — Итоговое содержимое файла lab9.xhtml проекта labs

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
      xmlns:f="http://xmlns.jcp.org/jsf/core" xml:lang="ru">

      <ui:composition template="/WEB-INF/templates/lab3Templ.xhtml">

      <!-- Переопределение контекстной страницы context -->
      <ui:define name="context">

      <div align="left" style="color:black;padding:10px">
            <b>lab9.xhtml - работа с сервисом:
            http://localhost:8080/lab9/letter</b>
            <hr/>
            <h:outputText value="#{lab9.resMsg}" style="color:red"/>
            <hr/>
            <h:form>
            <h:panelGrid columns="2">
                  <h:outputText value="Идентификатор:"/>
                  <h:inputText value="#{lab9.id}"/>

                  <h:outputText value="Введи текст:"/>
                  <h:inputTextarea value="#{lab9.text}" rows="5" cols="40"/>
            </h:panelGrid>

    <h:commandButton value="GET-список" action="#{lab9.mGetList()}"/>
    <h:commandButton value="GET-письмо" action="#{lab9.mGetLetter()}"/>
    <h:commandButton value="POST-новое" action="#{lab9.mPostLetter()}"/>
    <h:commandButton value="DELETE-письмо" action="#{lab9.mDeleteLetter()}"/>
    <h:commandButton value="PUT-письмо" action="#{lab9.mPutLetter()}"/>

      </h:form><hr/>
      <table id="table1" cellspacing="0" cellpadding="5" border="0" >
      <thead><tr>
            <th align="left" >№</th>
            <th align="left" >Дата</th>
            <th align="left" >Пользователь</th>
            <th align="left" >Сообщение</th>
      </tr></thead>

      <tbody>
      <c:forEach items="${lab9.list}" var="letter">
```

```
        <tr>
        <td><h:outputText value="${letter.id}" default="*"     /></td>
        <td><h:outputText value="${letter.dateString}"        /></td>
        <td><h:outputText value="${letter.name}"              /></td>
        <td><h:outputText value="${letter.text}"              /></td>
        </tr>
        </c:forEach>
        </tbody>
        </table>

        </div>
        </ui:define>
        </ui:composition>
</html>
```

Результат отображения ресурса, представленного данным листингом, показан на рисунке 6.14.

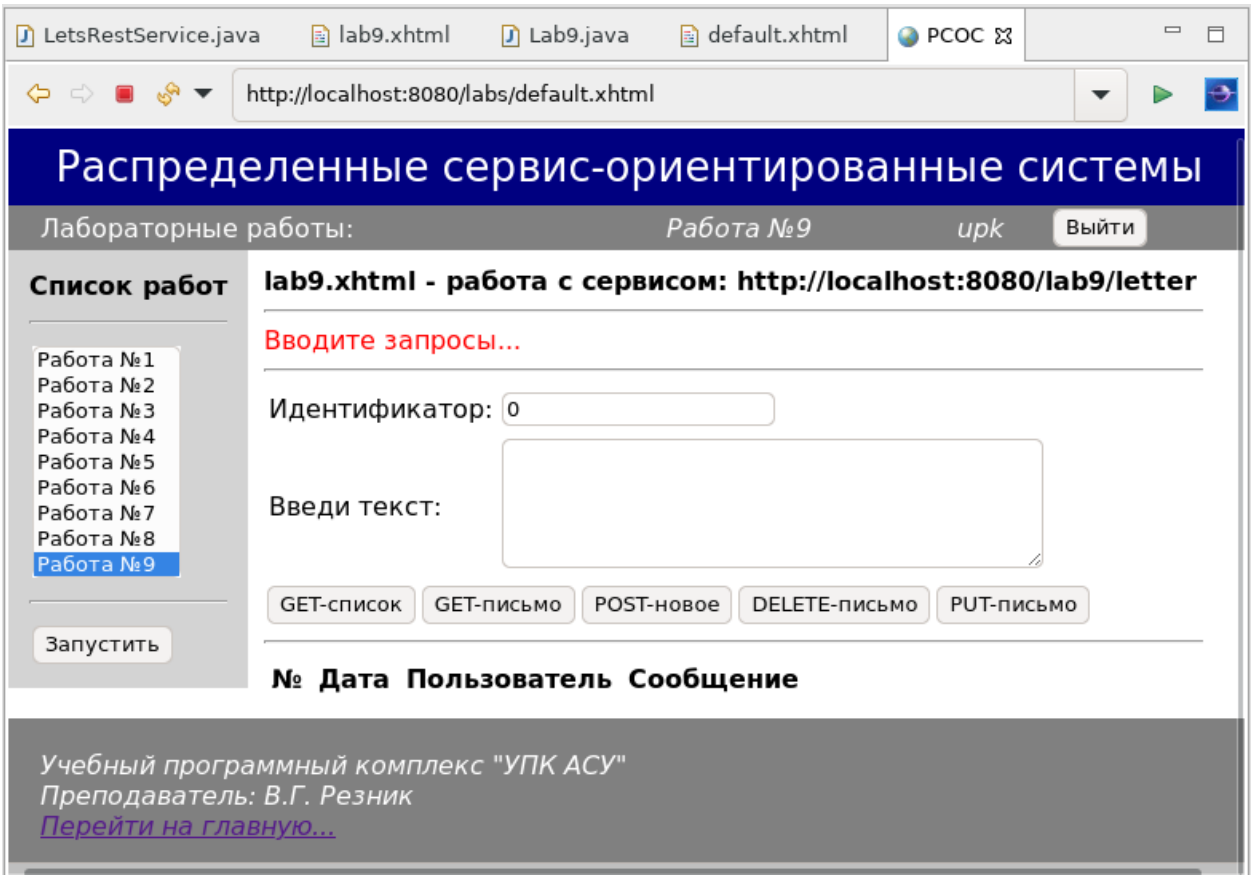

Рисунок 6.14 — Начальное изображение ресурса lab9.xhtml

Теперь перейдем к решению конкретных задач.

6.3.4 Клиент, реализующий методы GET и POST

Решение конкретных задач потребителей сервиса выполним посредством последовательной реализации методов запроса, обозначенных в шаблоне компоненты-подложки ***Lab9***. Данному пункту учебного материала соответствуют методы ***mGetList()***, ***mGetLetter()*** и ***mPostLetter()*** листинга 6.15.

Начнем последовательную реализацию рассматриваемой компоненты ***Lab9*** с метода ***mGetList()***, который должен:

1) ***проверить авторизацию*** пользователя и правильный выбор выполняемой работы, для чего следует воспользоваться методом ***isAuth()***; если авторизация не проходит, то выставить в приватную переменную соответствующее сообщение и вернуться назад к ресурсу ***lab9.xhtml***; если авторизация выполнена правильно, то выполнить следующие пункты;
2) ***сформировать*** и ***отправить запрос*** к производителю сервиса по указанному в переменной ***address*** значению адреса;
3) ***получить ответ*** в виде объекта типа ***Response***, извлечь из него код ответа и поместить этот код в переменную ***resMsg***; если код ответа не равен ***200***, то вернуться назад к ресурсу ***lab9.xhtml***;
4) ***извлечь*** из ***Response*** объект типа ***ListLets***;
5) ***извлечь*** из ***ListLets*** объект типа ***List<Letter>*** и присвоить его приватной переменной ***list***;
6) ***вернуться*** к ресурсу ***lab9.xhtml***.

Описанная выше реализация метода ***mGetList()*** показана на листинге 6.17.

Листинг 6.17 — Реализация метода mGetList() компоненты Lab9

```
//Получить список объектов типа Letter
public String mGetList()
{
        // Проверяем авторизацию
        if(!isAuth()) {
              resMsg = "Метод GET - нет авторизации!";
              return "lab9";
        }
        // Формируем запрос к сервису
        WebTarget target = ClientBuilder.newClient()
              .target(this.address);
        Invocation invocation =
                    target.request(MediaType.TEXT_PLAIN_TYPE)
                    .buildGet();

        // Отправка запроса и получение ответа
        Response response =
                    invocation.invoke();

        // Проверка кода возврата
```

```
            int status = response.getStatus();
            if(status != 200) {
            resMsg = "Метод GET - код возврата: "
                        + new Integer(status).toString() + "!";
            return "lab9";
            }
            resMsg = "Метод GET - код возврата: 200 - Хорошо";

            // Извлечение тела ответа в виде объекта типа ListLets
            ListLets body =
                        response.readEntity(ListLets.class);

            // Извлекаем список записей и возвращаемся к lab9.xhtml
            list = body.getList();
            return "lab9";
    }
```

После реализации метода ***mGetList()***, активация кнопки «***GET-список***» будет приводить к чтению всего списка записей и отображения его в окне ресурса, как это показано на рисунке 6.15.

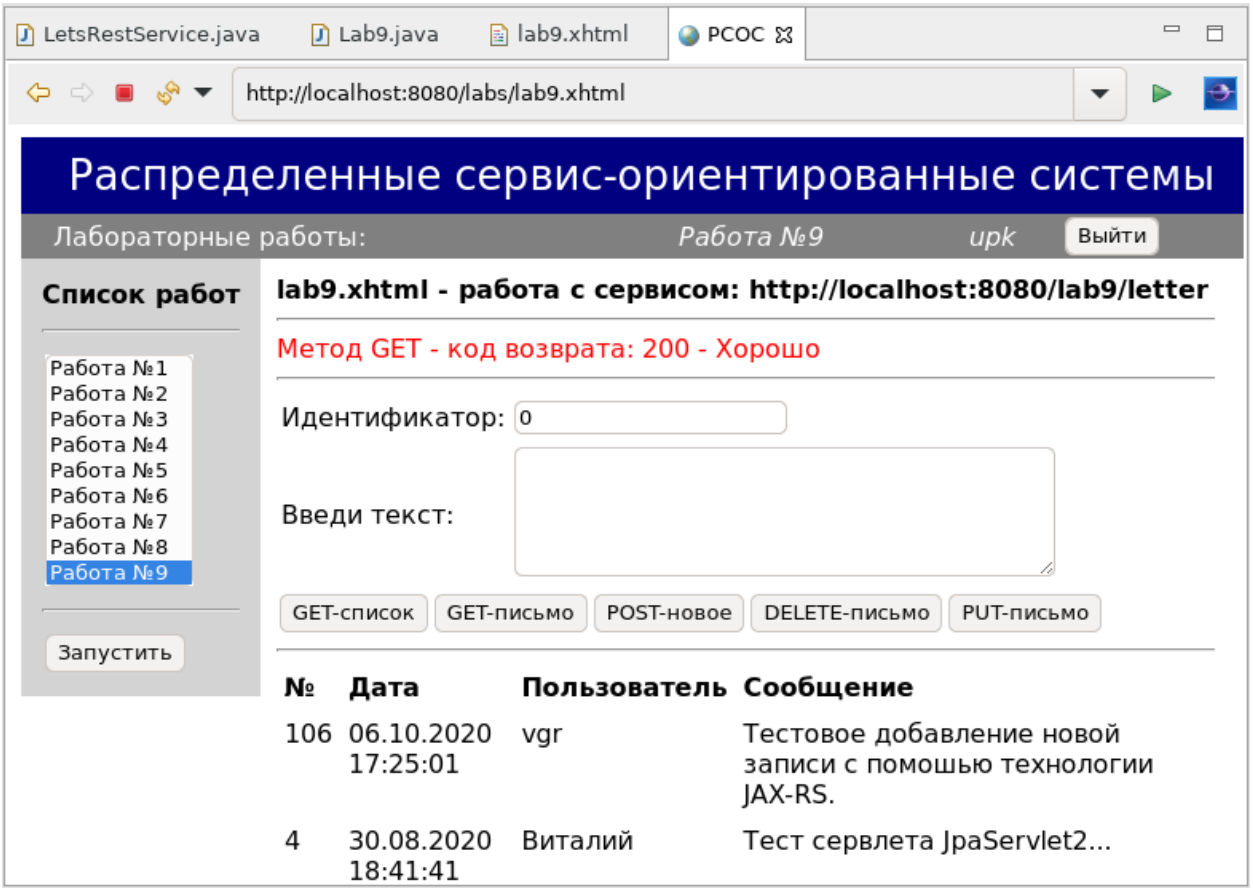

Рисунок 6.15 — Результат работы метода mGetList()

Теперь перейдем к реализации метода ***mGetLetter()***, который отличается от предыдущего метода следующими моментами:

а) при формировании адреса запроса, кроме переменной ***address*** используется также переменная ***id***;

б) если код возврата равен ***204***, то — адресуемый объект в ответе отсутсвует;

в) из объекта типа ***Response*** извлекается объект типа ***Letter***;

г) из объекта типа ***Letter*** извлекаются и устанавливаются переменные ***id*** и ***text***.

Описанная реализация метода ***mGetLetter()*** показана на листинге 6.18.

Листинг 6.18 — Реализация метода mGetLetter() компоненты Lab9

```
// Получить объект типа Letter по идентификатору id
public String mGetLetter() {
        // Проверяем авторизацию
        if(!isAuth()) {
              resMsg = "Метод GET - нет авторизации!";
              return "lab9";
        }
        // Формируем запрос к сервису
        WebTarget target = ClientBuilder.newClient()
              .target(this.address)
              .path(new Integer(id).toString());
        Invocation invocation =
                    target.request(MediaType.TEXT_PLAIN_TYPE)
                    .buildGet();

        // Отправка запроса и получение ответа
        Response response =
                    invocation.invoke();

        // Проверка кода возврата
        int status = response.getStatus();
        if(status == 204) {
        resMsg = "Метод GET - код возврата: "
                    + new Integer(status).toString()
                    + " - Отсутствует объект запроса с id="
                    + new Integer(id).toString();
        return "lab9";
        }
        if(status != 200) {
        resMsg = "Метод GET - код возврата: "
                    + new Integer(status).toString() + "!";
        return "lab9";
        }
        resMsg = "Метод GET - код возврата: 200 - Хорошо";

        // Извлечение тела ответа в виде объекта типа ListLets
        Letter body =
                    response.readEntity(Letter.class);

        // Извлекаем id, text и возвращаемся к lab9.xhtml
        id   = body.getId();
        text = body.getText();
        return "lab9";
```

```
}
```

После реализации метода ***mGetLetter()***, активация кнопки «***GET-письмо***» будет приводить к чтению отдельной записи и отображения ее в окне ресурса, как это показано на рисунке 6.16.

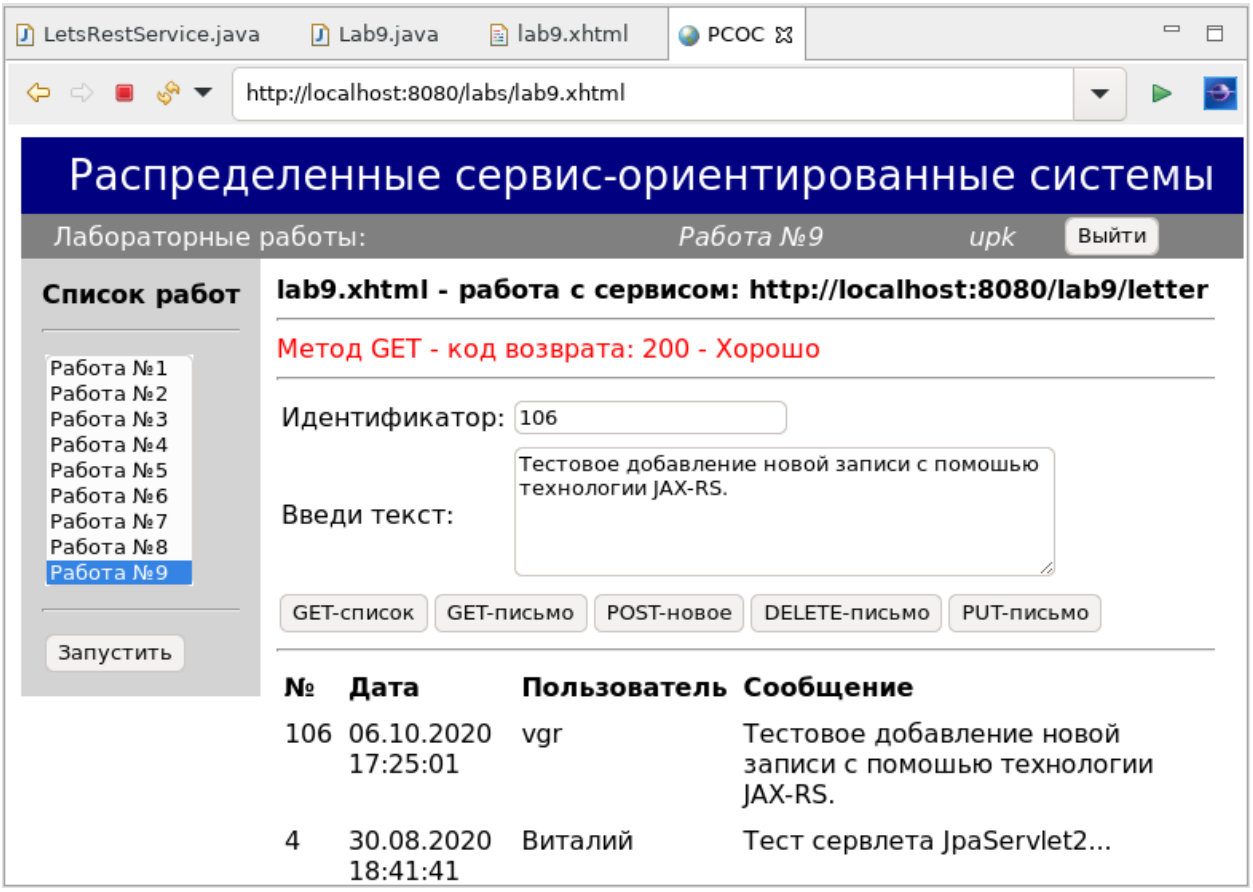

Рисунок 6.16 — Результат работы метода mGetLetter()

Теперь перейдем к следующей задаче. Она предполагает реализовать в компоненте-подложке ***Lab9*** метод ***mPostLetter()***. Для этого, должен быть создан объект типа ***Letter***, включающий:

а) ***заданное*** значение переменной ***text***;

б) ***сгенерированное*** значение типа ***Date***;

в) ***прочитанное*** из компоненты ***RSOS*** имя пользователя ***name***.

Обратите внимание, что переменная ***id*** не используется, поскольку она будет присвоена самой СУБД.

В самом запросе передается объект типа ***Letter***, а в ответе объекта типа ***Response*** возможны две ситуации:

1) код статуса ***201*** - Создано с URI нового ресурса;

2) код статуса ***204*** — Нет содержимого, если метод не создал ресурс.

Описанная выше реализация метода ***mPostLetter()*** представлена на листинге 6.19.

Листинг 6.19 — Реализация метода mPostLetter() компоненты Lab9

```
// Создать новый объект типа Letter
public String mPostLetter() {
      // Проверяем авторизацию
      if(!isAuth()) {
            resMsg = "Метод POST - нет авторизации!";
            return "lab9";
      }
      // Создаем новый объект типа Letter
      Letter letter =
                  new Letter(new Date(), rsos.getUser(), text);

      // Формируем запрос к сервису
      WebTarget target = ClientBuilder.newClient()
                  .target(this.address)
                  .register(Letter.class);
      Invocation invocation =
                  target.request(MediaType.APPLICATION_XHTML_XML)
      .buildPost(Entity.entity(letter, MediaType.APPLICATION_XML_TYPE));

      // Отправка запроса и получение ответа
      Response response =
                  invocation.invoke();

      // Проверка кода возврата
      int status = response.getStatus();
      if(status == 204) {
            resMsg = "Метод POST - код возврата: "
                        + new Integer(status).toString()
                        + " - Нет содержимого";
            return "lab9";
      }
      if(status != 201) {
            resMsg = "Метод POST - код возврата: "
                        + new Integer(status).toString() + "!";
            return "lab9";
      }
      // Извлечение адреса созданного объекта
      resMsg = "Метод POST - адрес сообщения: "
                     + response.getLocation();

      // Возвращаемся к lab9.xhtml
      return "lab9";
    }
```

После реализации метода ***mPostLetter()***, активация кнопки «***POST-новое***» будет приводить к созданию новой отдельной записи и отображению ее адреса в окне ресурса, как это показано на рисунке 6.17.

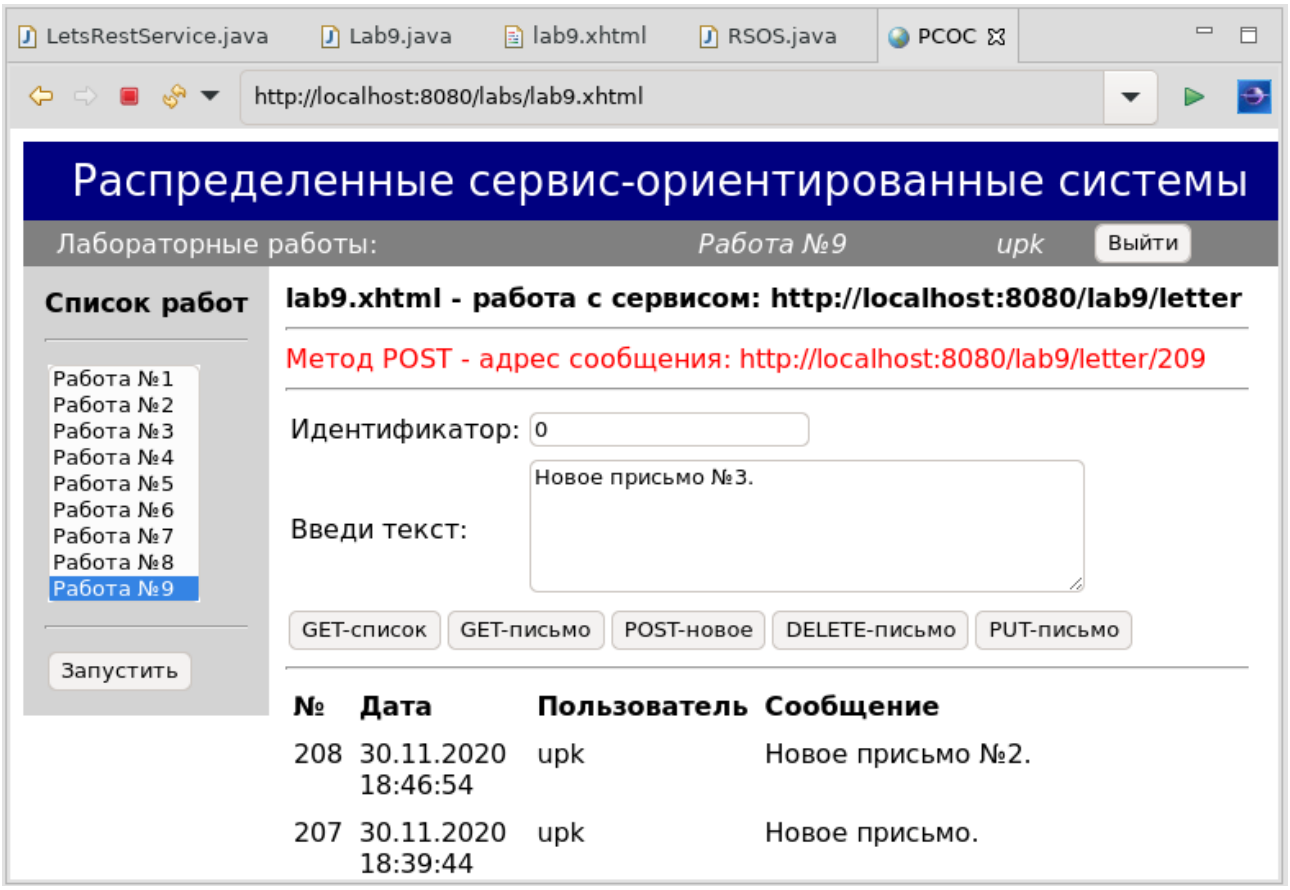

Рисунок 6.17 — Результат работы метода mPostLetter()

Чтобы новое сообщение появилось в общем списке, необходимо активировать кнопку «***GET-список***».

6.3.6 Клиент, реализующий методы DELETE и PUT

Реализация метода ***mDeleteLetter()*** во многом похожа на реализацию метода ***mGetLetter()***, поскольку при формировании адреса запроса, кроме переменной ***address*** используется также переменная ***id***. Кроме того, необходимо реализовать анализ трех кодов возврата:

а) если код возврата равен ***200***, то объект типа ***Response*** содержит удаленный объект и его необходимо извлечь, как описано ниже;

б) если код возврата равен ***202***, то действие по удалению объекта принято к исполнению, но еще не завершилось;

в) если код возврата равен ***204***, то адресуемый объект — удален, но в ответе его тело отсутствует;

При получении кода возврата ***200***, из объекта типа ***Response*** извлекается объект типа ***Letter***, а затем — из объекта типа ***Letter*** извлекаются и устанавливаются переменные ***id*** и ***text***.

Описанная реализация метода ***mDeleteLetter()*** показана на листинге 6.20.

Листинг 6.20 — Реализация метода mDeleteLetter() компоненты Lab9

```
    // Удалить объект типа Letter по идентификатору id
    public String mDeleteLetter() {
            // Проверяем авторизацию
            if(!isAuth()) {
                  resMsg = "Метод DELETE - нет авторизации!";
                  return "lab9";
            }
            // Формируем запрос к сервису
            WebTarget target = ClientBuilder.newClient()
                  .target(this.address)
                  .path(new Integer(id).toString());
            Invocation invocation =
                        target.request()
                        .buildDelete();

            // Отправка запроса и получение ответа
            Response response =
                        invocation.invoke();

            // Проверка кода возврата
            int status = response.getStatus();
            if(status == 202) {
            resMsg = "Метод DELETE - код возврата: "
                        + new Integer(status).toString()
                        + " - Запрос с id="
                        + new Integer(id).toString()
                        + " - принят к исполнению...";
            return "lab9";
            }
            if(status == 204) {
            resMsg = "Метод DELETE - код возврата: "
                        + new Integer(status).toString()
                        + " - Отсутствует объект запроса с id="
                        + new Integer(id).toString();
            return "lab9";
            }
            if(status != 200) {
            resMsg = "Метод DELETE - код возврата: "
                        + new Integer(status).toString() + "!";
            return "lab9";
            }
            resMsg = "Метод DELETE - код возврата: 200 - Хорошо";

            // Извлечение тела ответа в виде объекта типа Letter
            Letter body =
                        response.readEntity(Letter.class);

            // Извлекаем id, text и возвращаемся к lab9.xhtml
            id   = body.getId();
            text = body.getText();
            return "lab9";
    }
```

Для удаления записи необходимо ввести ее идентификатор и активировать кнопку «***DELETE-письмо***». Результат удаления записи с ***id=106*** показан на рисунке 6.18.

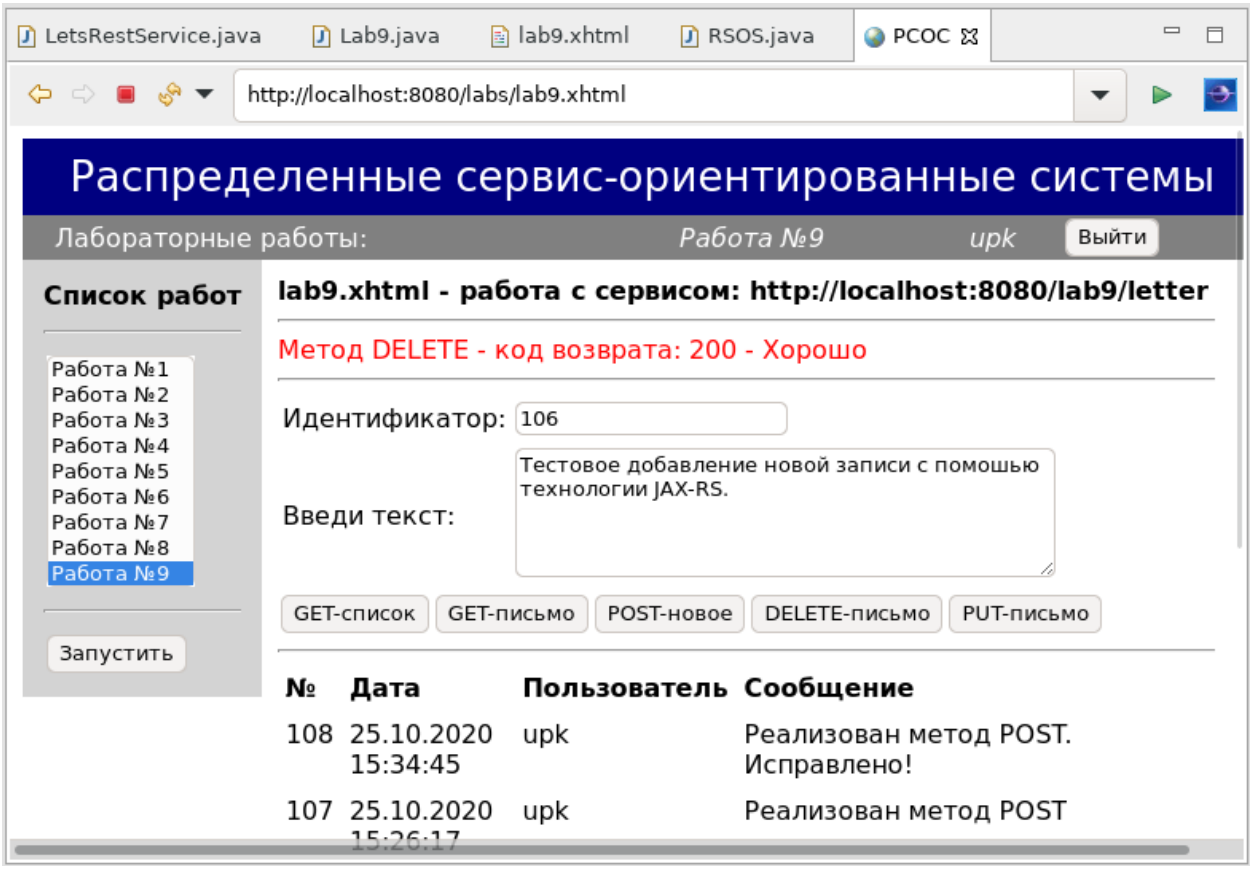

Рисунок 6.18 — Результат удаления записи методом mDeleteLetter()

Теперь перейдем к реализации метода ***mPutLetter()***.

Реализация метода ***mPutLetter()*** во многом похожа на реализацию метода ***mPostLetter()***, поскольку в обоих случаях осуществляется передача объекта типа ***Letter*** производителю сервиса. Что касается кодов статуса объекта типа ***Response***, то их — всего два:

1) ***код 200 — Хорошо***, если в теле ответа возвращен измененный объект;
2) ***код 204 — Нет содержимого***, если изменений не произошло.

При указанных условиях и ограичениях, описанная выше реализация метода ***mPutLetter()*** представлена на листинге 6.21.

Листинг 6.21 — Реализация метода mPutLetter() компоненты Lab9

```
// Модифицировать объект типа Letter по идентификатору id
public String mPutLetter() {
	// Проверяем авторизацию
	if(!isAuth()) {
		resMsg = "Метод PUT - нет авторизации!";
		return "lab9";
	}

	// Создаем новый объект типа Letter
	Letter letter =
			new Letter(new Date(), rsos.getUser(), text);
	letter.setId(id);

	// Формируем запрос к сервису
	WebTarget target = ClientBuilder.newClient()
			.target(this.address)
			.register(Letter.class);
	Invocation invocation =
		target.request(MediaType.APPLICATION_XHTML_XML)
		.buildPut(Entity.entity(letter, MediaType.APPLICATION_XML_TYPE));

	// Отправка запроса и получение ответа
	Response response =
			invocation.invoke();

	// Проверка кода возврата
	int status = response.getStatus();
	if(status == 204) {
		resMsg = "Метод PUT - код возврата: "
				+ new Integer(status).toString()
				+ " - Нет содержимого";
		return "lab9";
	}
	if(status != 200) {
		resMsg = "Метод PUT - код возврата: "
				+ new Integer(status).toString() + "!";
		return "lab9";
	}

	resMsg = "Метод PUT - код возврата: 200 - Хорошо";

	// Извлечение тела ответа в виде объекта типа Letter
	Letter body =
		response.readEntity(Letter.class);

	// Извлекаем id, text и возвращаемся к lab9.xhtml
	id   = body.getId();
	text = body.getText();
	return "lab9";
}
```

Результат модификации записи с ***id=109*** показан на рисунке 6.19.

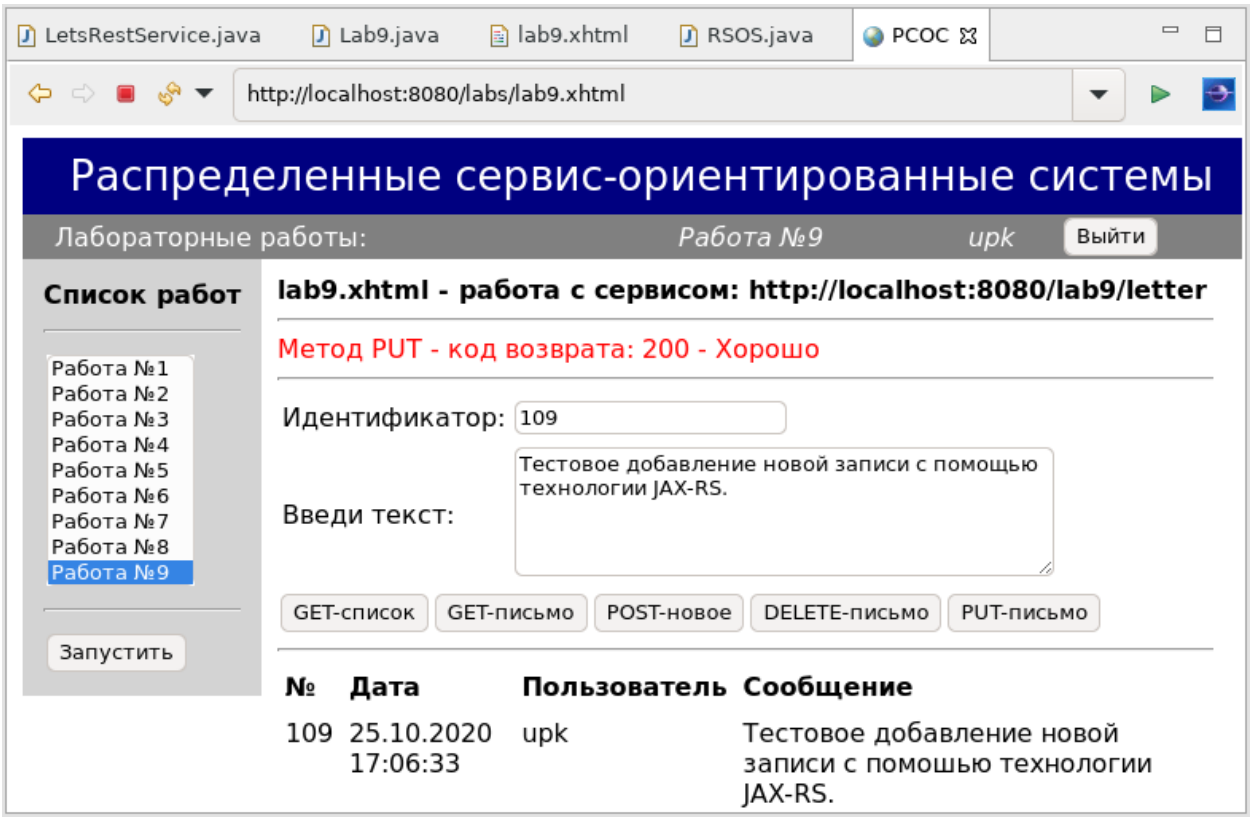

Рисунок 6.19 — Результат модификации записи методом mPutLetter()

Обратите внимание, что после обновления записи и, если оно прошло успешно, то обновленный вариант переписывается в форму, а список всех записей остается неизменным. Чтобы изменить список всех записей, необходимо активировать кнопку «***PUT-письмо***».

Данным примером, мы завершаем изучение учебного материала данной главы и дисциплины в целом. Надеюсь, что полученные в процессе обучения навыки и знания обеспечат дальнейший профессиональный рост студентов.

Вопросы для самопроверки

1. Что понимается под обозначением REST?
2. Что называется RESRful-системами?
3. Что понимается под обозначением CRUD?
4. Кто и когда предложил термин REST?
5. Сколько и какие ограничения необходимы для того, чтобы приложение называлось REST-системой?
6. Приведите определение понятию «*Ресурс*».
7. Что понимается под термином «*Адресуемость*»?
8. Какие методы доступа к Web-сервису соответствуют обозначению CRUD?
9. Что такое - «*Согласование содержимого*»?
10. Какие группы кодов возвращает Web-сервис потребителю сервиса?
11. Каково назначение языков WADL и HAL?
12. Какие аннотации необходимы, чтобы JAVA-класс воспринимался как RESTful-сервис?
13. Какими аннотациями обозначаются основные методы доступа поставщика RESTful-сервиса?
14. Какие аннотации параметров запроса вы знаете?
15. Какие семь классов Java необходимо знать, чтобы реализовывать клиента RESTful-сервиса?

Заключение

Завершив описание данной дисциплины, следует подвести итоги по представленному в ней учебному материалу, чтобы студент мог оценить конечный результат, полученный в результате его изучения.

По замыслу исполнения, представленный учебный материал условно разделяется на три части, каждая из которых на своем уровне раскрывает тематику распределенных сервис-ориентированных систем (РСОС).

Первая условная часть отражена в первой главе учебного пособия, где представлено описание пяти наиболее значимых факторов, существенно повлиявших на формирование как самого научного направления сервис-ориентированных систем, так и на организационную часть учебного процесса по изучению этого направления. И хотя эти факторы действуют комплексно, но достаточно самостоятельно, поэтому они упорядоченны автором по степени своей познавательной значимости, начиная с исторической преемственности РСОС от моделей распределенных операционных систем до моделей объектного подхода, реализованных в технологии CORBA. Далее показано влияние на теоретические представления о РСОС фактора Web-технологий и достаточно прагматического фактора масштабирования приложений на уровень моделей предприятия. После этого, по результатам краткого обощения достаточно разнородных теоретических обобщений, становится понятным выбор инструментальных средств создания РСОС, который ориентируется на контейтерные технологии (компонентный и аспектно-ориентированный подходы), реализуемые серверами приложений. Конкретизация такого выбора ложится на плечи программной платформы Java EE и непосредственного набора открытых (open) программных продуктов, описанных в подразделах 1.4 и 1.5 этой главы.

Вторая условная часть изучаемой дисциплины описана в последующих трех главах, посвященных наиболее современным инструментальным средствам программной платформы Java EE. Без изучения этих инструментальных средств, невозможно обеспечить качественное освоение изучаемого предмета. Действительно, во второй главе дается описание технологии JSF, которая не только обеспечивает представление информации для потребителя сервиса, но и наиболее полно концентрирует в себе все технологические достижения управляемых контейнеров (CDI), реализованных в современных серверах приложений. Это создает возможность в третьей главе изучить технологию современных средств работы с базами данных, концентрируя внимание студента на ORM-отображении объектов-сущностей в базы данных, управляемых классическими средствами СУБД, основанными на языке SQL. Дополнительно, здесь рассматриваются и средства создания EJB-компонент, на основе которых в последующем и создаются сервисные системы. Наконец, содержимое четвертой главы завершает изучение инструментальных средств, описанием технологий

JAXB и JSON.

Третья условная часть изучаемой дисциплины описана в последних двух главах. Здесь собственно и показаны примеры реализации сервис-ориентированных систем. Так в главе 5, описан учебный пример, который реализует классическую Web-службу, использующую протокол SOAP. Аннотации этой технологии, разработанные в последних версиях проекта Java EE, обеспечивают оптимальную для программиста реализацию таких систем.

Что касается главы 6, то она дает общее представление о современном, альтернативном классическому, подходе, призванному использовать стиль проектирования и реализации REST. По разным оценкам этот стиль позволяет реализовывать более эффективные PCOC-системы, чем классический подход. Здесь студенту предоставляется полная реализация учебного примера, включающая использование технологий JSF, EJB-компонент и специально аннотированного сервлета, соответствующего стилю REST.

Подводя итог изложенному учебному материалау, следует отметить и перечень вопросов, не вошедших в программу учебного курса по причине ограниченности его временных рамок. К ним относятся:

1) вопросы валидации компонент программного обеспечения, позволяющими справляться с регулярно возникающими проблемами внедрения зависимостей, контроля вводимой пользователем информации, созданием перехватчиков и управлением жизненным циклом компонент;
2) многие вопросы обработки функций обратного вызова, использование служб таймеров и достаточно обширная тема различных аспектов обеспечения безопастности;
3) вопросы асинхронного взаимодействия слабосвязанных компонент сервис-ориентированных систем;
4) вопросы инструментального использования языка JavaScript и популярной сейчас технологии AJAX.

В целом, перечень не расмотренных в данной дисциплине вопросов можно продолжить, например, широко популярная в JAVA-сообществе технология **Spring Framework**, появившаяся в октябре 2002 года, рассматривается многими разработчиками как реальная конкурирующая альтернатива программной платформе Java EE.

Автор данного пособия осознает ограниченность представленного здесь учебного материала, но считает данную ситуацию вынужденной необходимой мерой, призванной сформировать у студента прочные познавательные основы. Используя эти основы, студент может продолжить свое обучение и найдет, а может и создаст, свои более совершенные инструменты реализации распределенных сервис-ориентированных систем.

Список использованных источников

1. Резник В. Г. Распределенные вычислительные сети: Учебное пособие [Электронный ресурс] / В. Г. Резник. — Томск: ТУСУР, 2019. — 211 с. — Режим доступа: https://edu.tusur.ru/publications/9072.
2. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем: Учебник для вузов. 3-е изд. — СПб.: Питер, 2015. — 688 с.: ил. (Серия «Учебник для вузов»). - ISBN 978-5-496-01145-7.
3. Ларионов А. М., Майоров С.А., Новиков Г. И. ВЫЧИСЛИТЕЛЬНЫЕ КОМПЛЕКСЫ, СИСТЕМЫ И СЕТИ. - Ленинград, ЭНЕРГОАТОМИЗДАТ, 1987. - 178 с.
4. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. — СПб.: Питер, 2003. — 877 с.: ил. — (Серия «Классика computer science»). - ISBN 5-272-00053-6.
5. SOA Архитектурные особенности и практические аспекты [Электронный ресурс] / TADVISER - 2010. — Режим доступа: http://www.tadviser.ru/index.php/Статья:SOA_Архитектурные_особенности_и_практические_аспекты.
6. Simple Object Access Protocol (SOAP) 1.1 [Электронный ресурс] / W3C - 2007. — Режим доступа: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
7. SOAP Version 1.2 Part 0: Primer (Second Edition) [Электронный ресурс] / W3C - 2007. — Режим доступа: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/.
8. Web Sevices Description Language (WSDL) 1.1 [Электронный ресурс] / W3C - 2001. — Режим доступа: https://www.w3.org/TR/2001/NOTE-wsdl-20010315.
9. Web Sevices Description Language (WSDL) Version 2.0 Part 1: Core Language [Электронный ресурс] / W3C - 2007. — Режим доступа: https://www.w3.org/TR/wsdl20/.
10. UDDI Version 3.0.2 [Электронный ресурс] / OASIS - 2004. — Режим доступа: http://www.uddi.org/pubs/uddi-v3.0.2-20041019.htm.
11. Радченко, Г.И. Распределенные вычислительные системы / Г.И. Радченко. – Челябинск: Фотохудожник, 2012. — 184 с. — ISBN 978-5-89879-198-8.
12. Reference Model for Service Oriented Architecture 1.0 [Электронный ресурс] / OASIS - 2006. 33 с. — Режим доступа: https://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf.

13. Гриценко Ю. Б.Архитектура предприятия: учеб. пособие / Ю.Б. Гриценко. — Томск: Изд-во Томск. гос. ун-та систем управления и радиоэлектроники, 2014. — 260 с. ISBN 978-5-86889-512-8.
14. Глод, О.Д. Архитектура предприятия: учебное пособие / О.Д. Глод; Южный федеральный университет.– Таганрог: Издательство Южного федерального университета, 2016. – 93 с.
15. Р 50.1.041-2002 РУКОВОДСТВО ПО ПРОЕКТИРОВАНИЮ ПРОФИЛЕЙ СРЕДЫ ОТКРЫТОЙ СИСТЕМЫ (СОС) ОРГАНИЗАЦИИ-ПОЛЬЗОВАТЕЛЯ. - М.: ИПК Издательство стандартов, 2003. - 39 с.
16. Дашнер С. Изучаем Java EE. Современное программирование для больших предприятий. — СПб.: Питер, 2018. — 384 с.: ил. — (Серия «Для профессионалов»). ISBN 978-5-4461-0774-2.
17. Гонсалвес Э. Изучаем Java EE 7. — СПб.: Питер, 2014. — 640 с.: ил. ISBN 978-5-496-00942-3.
18. Шилдт Г. Java 8. Полное руководство; 9-е изд.: Пер. с англ. - М.: ООО "И.Д. Вильяме", 2015. - 1 376 с. : ил. - Парал. тит. Англ. ISBN 978-5-8459-1918-2 (рус.).
19. JSR 365: Contexts and Dependency Injection for Java 2.0 [Электронный ресурс] / Red Hat, Inc. - April 20th 2017. — Режим доступа: https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html.
20. Машнин Т. С. Web-сервисы Java. — СПб.: БХВ-Петербург, 2012. — 560 с.: ил. — (Профессиональное программирование) ISBN 978-5-9775-0778-3.
21. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux [Электронный ресурс]: Учебно-методическое пособие для студентов направления 09.03.01, направление подготовки "Программное обеспечение средств вычислительной техники и автоматизированных систем" / В.Г. Резник. — Томск: ТУСУР, 2016. — 33 с. — Режим доступа: https://edu.tusur.ru/publications/6238.
22. Apache Derby [Электронный ресурс]: Режим доступа: http://db.apache.org/derby/.
23. Apache TomEE [Электронный ресурс]: Режим доступа: http://tomee.apache.org/.
24. Eclipse Foundation [Электронный ресурс]: Режим доступа: https://www.eclipse.org/.
25. OmniFaces. To make JSF life easier [Электронный ресурс]: Режим доступа: https://omnifaces.org/.
26. JAXB Release Documentation [Электронный ресурс]: Режим доступа:

https://javaee.github.io/jaxb-v2/doc/user-guide/index.html.

27. Hypertext Transfer Protocol (HTTP) Status Code Registry [Электронный ресурс]: Режим доступа: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.
28. RFC 5988 — Web Linking [Электронный ресурс]: Режим доступа: https://tools.ietf.org/html/rfc5988.
29. JSON Hypermedia API Language [Электронный ресурс]: Режим доступа: https://tools.ietf.org/id/draft-kelly-json-hal-03.txt.
30. JAX-RS: Java™ API for RESTfulWeb. ServicesVersion 2.0 Final ReleaseMay 22, 2013 [Электронный ресурс]: Режим доступа: https://download.oracle.com/javaee-archive/jax-rs-spec.java.net/jsr339-experts/att-3593/spec.pdf.

Алфавитный указатель

С

Т

У

Э

A

B

C

S

U

V

W

X

Оглавление