ТЕХНОЛОГИИ ВЕБ-СЕРВИСОВ

Материалы для подготовки к выполнению лабораторных работ и экзамену

Дергачев А.М., Сафронов А.Г.

Санкт-Петербург

2014

Оглавление

Введение 4

Глава 1. Обзор сервис-ориентированной архитектуры 5

1.1. Сервис 5

1.2. Стандарты сервис-ориентированной архитектуры 6

1.2.1. SOAP 6

1.2.2. WSDL 8

1.2.3. UDDI 11

1.3. Взаимодействие сторон 12

1.4. Системы управления сервисами 13

1.4.1. Поисковый реестр 16

1.4.2. Системы анализа качества обслуживания 16

1.4.3. Сервисные шины предприятия 17

1.4.4. Обзор системы управления сервисами WSMX 20

Глава 2. Технологии разработки SOAP веб-сервисов 24

2.1. Процесс разработки 24

2.2. Обзор JAX-WS 25

2.2.1. wsgen 26

2.2.2. wsimport 27

2.3. Реализация веб-сервиса 28

2.3.1. Подготовка 28

2.3.2. Реализация сервиса 29

2.4. Запуск сервиса и просмотр его описаний 33

2.5. Тестирование сервиса с помощью SoapUI 34

2.6. Разработка клиента веб-сервиса 36

2.7. Разработка J2EE веб-сервиса 39

2.8. Обработка ошибок 42
2.8.1. Пример реализации обработки ошибок 43
Глава 3. REST-архитектура 48
3.1. Введение в REST 48
3.1. Введение в JAX-RS 52
3.2. Реализация сервиса с помощью JAX-RS 52
3.3. Реализация клиента 57
3.4. Реализация J2EE REST-сервиса 59
3.5. Обработка ошибок 62
Глава 4. SOAP или REST сервисы? 66
Глава 6. Реестры сервисов 69
6.1. UDDI-реестры 69
6.2. ebXML 71
6.3. Типы реестров 73
6.4. Существующие реестры 74
6.5. Apache jUDDI 77
6.5.1. Установка и запуск 77
6.5.2. Регистрация бизнеса 78
6.5.3. Регистрация сервиса 80
6.5.4. Поиск 81
6.5.5. Программное взаимодействие с jUDDI 81
Глава 5. Задания для самостоятельного выполнения 89
5.1. Работа №1. Поиск с помощью SOAP-сервиса. 89
5.2. Работа №2. Реализация CRUD с помощью SOAP-сервиса. 89
5.3. Работа №3. Обработка ошибок в SOAP-сервисе. 89
5.4. Работа №4. Поиск с помощью REST-сервиса. 90

5.5. Работа №5. Реализация CRUD с помощью REST-сервиса. 90
5.6. Работа №6. Обработка ошибок в REST-сервисе. 90
5.7. Работа №7. Регистрация и поиск сервиса в реестре jUDDI 90
Приложение 91
Maven 91
Необходимые для сборки зависимости Maven 92
Генерация артефактов веб-сервиса в IDE NetBeans 94
Настройка подключения к базе данных в GlassFish 95
Ссылки и рекомендуемая литература 96

Введение

В наши дни наблюдается устойчивый рост интереса к концепции сервис-ориентированной архитектуры. Свидетельство тому - оценки аналитических компаний и усилия крупных поставщиков программного обеспечения по продвижению этого подхода. Например, аналитики компании Gartner Group прогнозировали, что к 2008 году преобладающей практикой проектирования и разработки корпоративных систем и приложений станет сервис-ориентированная архитектура, а уже к 2010-му году, по меньшей мере, 65% крупных компаний переведут более 35% своего портфеля приложений на сервис-ориентированную архитектуру. Специалисты компании ZapThink также давали похожие прогнозы.

Но почему сервис-ориентированная архитектура подвергается такой тенденции? Из-за чего ее использование кажется таким привлекательным? Каковы ее основные принципы и какова ее роль в корпоративных системах?

Для того чтобы ответить на эти вопросы, в данном пособии будут представлены обзор сервис-ориентированной архитектуры, а также технологии разработки веб-вервисов с использованием платформы Java.

Глава 1. Обзор сервис-ориентированной архитектуры

Специалисты корпорации IBM дают следующее определение сервис-ориентированной архитектуры: «Сервис-ориентированная архитектура — это прикладная архитектура, в которой все функции определены как независимые сервисы с вызываемыми интерфейсами. Обращение к этим сервисам в определенной последовательности позволяет реализовать тот или иной бизнес-процесс». В то же время, с точки зрения разработки программного обеспечения, сервис-ориентированная архитектура — это компонентная модель, в которой различные функциональные единицы приложений, называемые сервисами, взаимодействуют друг с другом посредством интерфейсов.

1.1. Сервис

Фактически, сервис представляет собой набор логически связанных методов, которые могут быть программно вызваны, а результат работы сервиса – реализация определенной функции бизнес-логики. В процессе своей работы сервис может обращаться к другим сервисам.

Функциональность сервиса определяется его интерфейсом - в интерфейсе определены его методы, передаваемые в них параметры и возвращаемые результаты**.** На архитектурном уровне при обращении к сервису не имеет значения, является он локальным, то есть, реализован в данной системе или удаленным, то есть, внешним по отношению к ней, какой протокол используется для передачи вызова, какие компоненты инфраструктуры при этом задействованы. SOA предполагает наличие единой схемы обращения к сервису независимо от того, где он находится. Интеграция сервисов осуществляется с помощью известных стандартных протоколов. Чаще всего таким протоколом является SOAP. Благодаря этому сервисы можно разрабатывать, используя разные платформы, и в одном сервисе осуществлять вызов сервисов, реализованных на разных платформах.

Фактически, если вся система целиком состоит из совокупности сервисов, то можно сказать, что сервисы в данном случае – это своего рода “кирпичи”, из которых собирается эта система. С точки зрения бизнес-процессов, сервисы – это

определенные “единицы работы”, используемые для реализации функций бизнес-логики.

Зачастую сервис не является отдельным самостоятельным приложением, а функционирует в составе сервера приложений, выбор которого обусловлен платформой, используемой для разработки этого сервиса.

Для того чтобы воспользоваться услугами сервиса к нему необходимо сделать запрос. Следовательно, во взаимодействии, в котором участвует сервис можно выделить две стороны: клиента – потребителя услуг сервиса и, непосредственно, саму серверную часть, обслуживающую клиентские запросы. Важно отметить, что, фактически, клиентом является программный модуль, осуществляющий формирование и отправку запроса по протоколу SOAP, а также интерпретирующий ответное на этот запрос сообщение.

1.2. Стандарты сервис-ориентированной архитектуры

Теперь, после того как приведены первоначальные сведения об основных рабочих программных модулях сервис-ориентированной архитектуры – о сервисах, необходимо показать основные стандарты, регламентирующие взаимодействие в рамках этой архитектуры, а также описать процесс взаимодействия.

Принято считать, что SOA базируется на трех основных стандартах: SOAP, WSDL и UDDI. Первый из них, который будет рассмотрен - SOAP.

1.2.1. SOAP

SOAP (Simple Object Access Protocol) – протокол обмена структурированными сообщениями в формате XML, работающий поверх протокола HTTP. Любое взаимодействие клиента с сервисом осуществляется по этому протоколу. Символьные данные могут передаваться в теле сообщения, а двоичные – прикрепляться к самому сообщению.

Фактически сообщение протокола SOAP представляет собой конверт, в который вложены заголовок сообщения и тело сообщения.

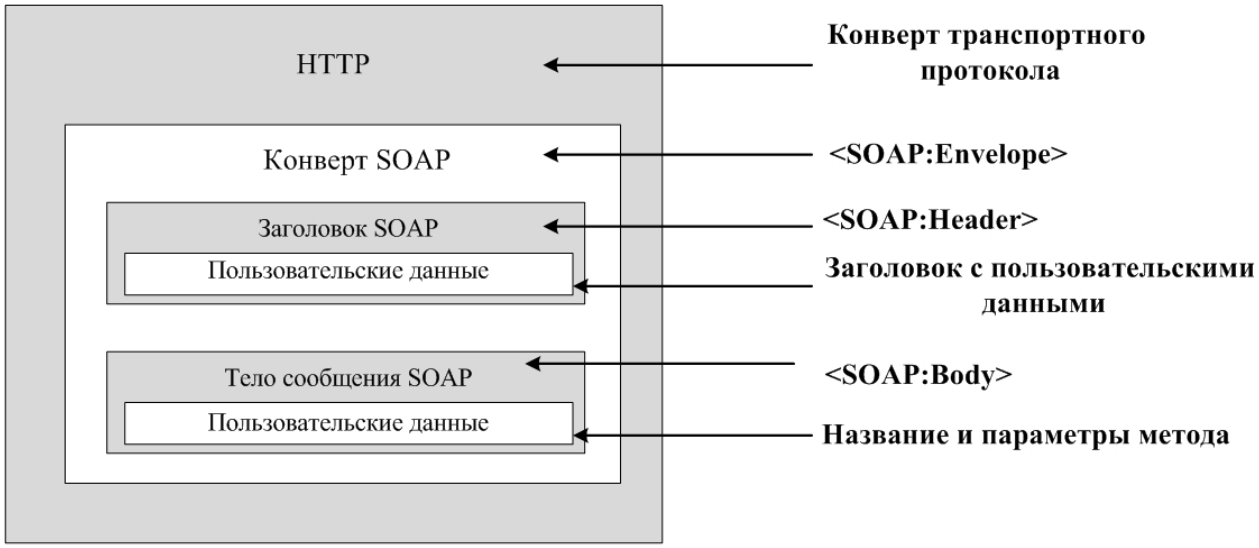


Рис. 1.1. SOAP-сообщение

Сообщение протокола SOAP имеет следующую струкруту

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Header>

...

Управляющая информация относительно всего SOAP-сообщения

...

</soap:Header>

<soap:Body>

...

Пользовательские данные

...

</soap:Body>

<soap:Fault>

...

Информация об ошибках

...

</soap:Fault>

</soap:Envelope>

Корневой тег SOAP-сообщения – тег Envelope. Он содержит в себе заголовок, тело сообщения, а также информацию об ошибках. В заголовке может находиться определенная управляющая информация, которая будет проанализирована при разборе сообщения на серверной или клиентской стороне. В теле сообщения содержатся данные запроса или ответа. В теге Fault содержатся сведения о произошедших ошибках.

Далее приведен пример сообщения протокола SOAP:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://example.com/ws">
      <productID>12345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Данное сообщение отправляется сервису клиентом, когда клиент осуществляет вызов метода getProductDetails (тег getProductDetails – единственный тег в теле сообщения), при этом значение входного параметра productID равно “12345”.

Благодаря использованию протокола SOAP сервис-ориентированной архитектуре удается достичь универсальности потому, что при передаче сообщений объекты передаются в универсальном, стандартизированном формате XML. Но также из-за этого значительно увеличивается размер сообщения, и снижается скорость его обработки.

1.2.2. WSDL

Второй стандарт, который будет рассмотрен: WSDL - язык описания веб-сервисов. С точки зрения синтаксиса WSDL – это XML-подобный язык. Каждый сервис имеет свое описание на языке WSDL, которое позволяет его однозначно охарактеризовать. Такое описание можно назвать контрактом, который клиент использует для того чтобы узнать некоторые подробности про сервис, а также принять решение о том, совершать ли к нему обращение или нет.

В таких контрактах определяются имя, адрес сервиса, типы входных и выходных данных, сообщения, используемые сервисом, список методов сервиса, особенности доставки сообщений и другие технические особенности сервиса. Фактически, в этом контракте описаны интерфейс сервиса, а информация о политиках сервиса, определяющих условия взаимодействия с ним.

Можно сказать, что WSDL позволяет описать сервис на двух уровнях: на абстрактом и детальном. На абстрактном уровне описывается интерфейс сервиса, который определяет его операции, и, как следствие, сообщения сервиса. То есть этот уровень служит для описания того, “что” должен делать сервис. В то же время важно не только “что” должен делать сервис, но и “как” это делать. Именно для этого и служит описание детального уровня. Тут уже определяются детали протокола обмена сообщениями, формат сообщений, адрес сервиса и другие параметры. В Приложении №2 приведено описание структуры документа WSDL.

Документ WSDL имеет следующую структуру:

```
<definitions>
   <types>
          Определение типов данных
   </types>
   <message>
          Определение входных и выходных сообщений
   </message>
   <portType>
          Определение портов
   </portType>
   <binding>
          Определение связей
   </binding>
</definitions>
```

Корневым тегом документа WSDL является тег <definitions>. В данный документ входит описание типов входных и выходных данных, сообщений, портов и связей.

Типы данных, используемые сервисом, описываются в элементе <types>.

Элемент <message> описывает структуру входных и выходных сообщений сервиса. Сообщение может состоять из нескольких частей. Части сообщения описываются в этом же элементе.

В элементе <portType> описываются операции сервиса, то есть именно здесь определяется его функциональность. Операции описываются в элементах <operation>. Как правило, операция характеризуется именем, входным и выходным сообщением.

Связи описываются в элементе <binding>. Связи определяют настройки протокола SOAP для каждого из портов.

Далее приведен пример документа WSDL:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
  </operation>
</portType>
```

Данный WSDL описывает веб-сервис, у которого есть один метод getTermRequest, который возвращает объект getTermResponse. Также в данном документе описаны сообщения getTermRequest и getTermResponse. Оба эти сообщения имеют по одному строковому полю. Подобное описание передаваемых данных в документе WSDL представлено в виде XSD-схемы.

XSD (XML Schema definition) – язык описания структуры XML-документа. Применительно к веб-сервисам, XSD описывает требуемую структуру входных и выходных сообщений сервиса.

Далее приведен пример XSD-схемы:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name='country'>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name"type="xs:string"/>
```

```
        <xs:element name="population" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Данная схема описывает xml, корневой тег которого называется country. Также в тег country вложены тег name, который имеет строковое значение и тег population, значение которого представлено числом с плавающей запятой.

Например, данный xml соответствует данному описанию:

```
<?xml version="1.0" encoding="utf-8"?>
<country>
  <name>United States of America</name>
  <population>340.4</population>
</country>
```

1.2.3. UDDI

UDDI (Universal Description, Discovery and Integration) – стандарт для создания особого реестра, позволяющего публиковать и находить сервисы. Фактически, UDDI-реестр является специализированным приложением, используя функциональность которого, поставщики сервисов регистрируют свои сервисы, а потенциальные клиенты ищут наиболее подходящие для них. Как правило, такие системы имеют возможности классификации и категоризации зарегистрированных в них сервисов.

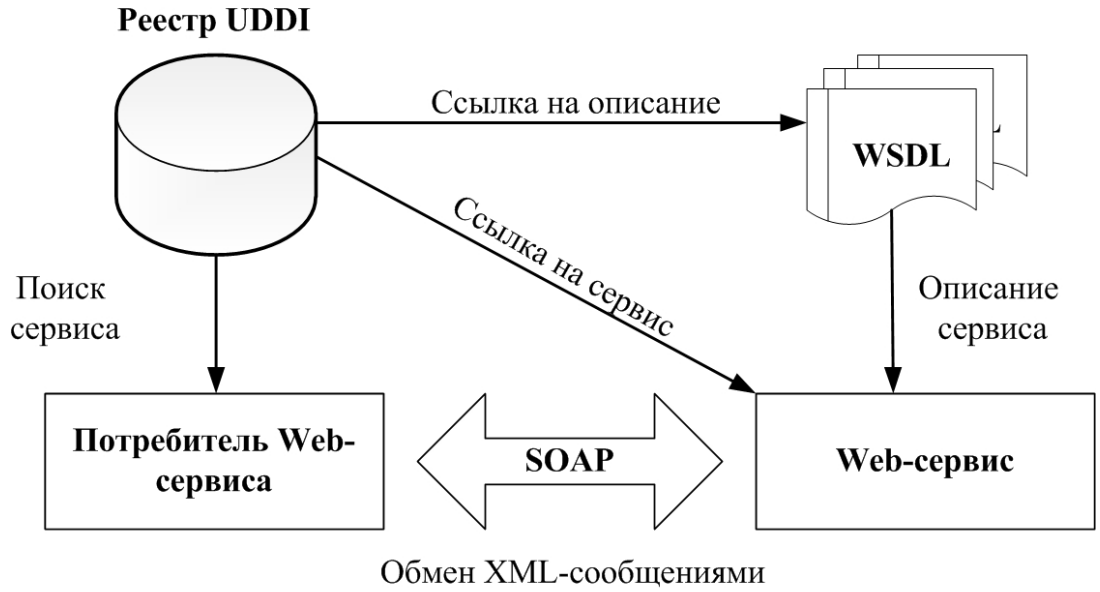


Рис. 1.2. Организация реестра UDDI

Важно отметить, что информацию из реестра можно условно разделить на три группы по характеру данной информации: белые страницы, желтые страницы и зеленые страницы. Белые страницы характеризую поставщика сервиса, предоставляя его описание, контактные данные и другую личную информацию. Желтые страницы описывают сервисы по сферам применения, а также каталогизируют и классифицируют их. Зеленые страницы несут в себе техническую информацию, необходимую для обращения к сервисам.

Таким образом, поиск можно проводить по различным критериям, начиная от местоположения поставщика услуг и, заканчивая особенностями интерфейса сервиса.

1.3. Взаимодействие сторон

В самом общем виде сервис-ориентированная архитектура предполагает наличие трех основных участников: поставщика сервиса, потребителя сервиса и реестра сервисов. Взаимодействие участников выглядит следующим образом: поставщик сервиса регистрирует свои сервисы в реестре, а потребитель обращается к реестру с запросом и выбирает наиболее подходящий для него сервис. После этого клиенту остается только обратиться к сервису. Все взаимодействия сторон происходят с использованием вышеописанных стандартов и протоколов.

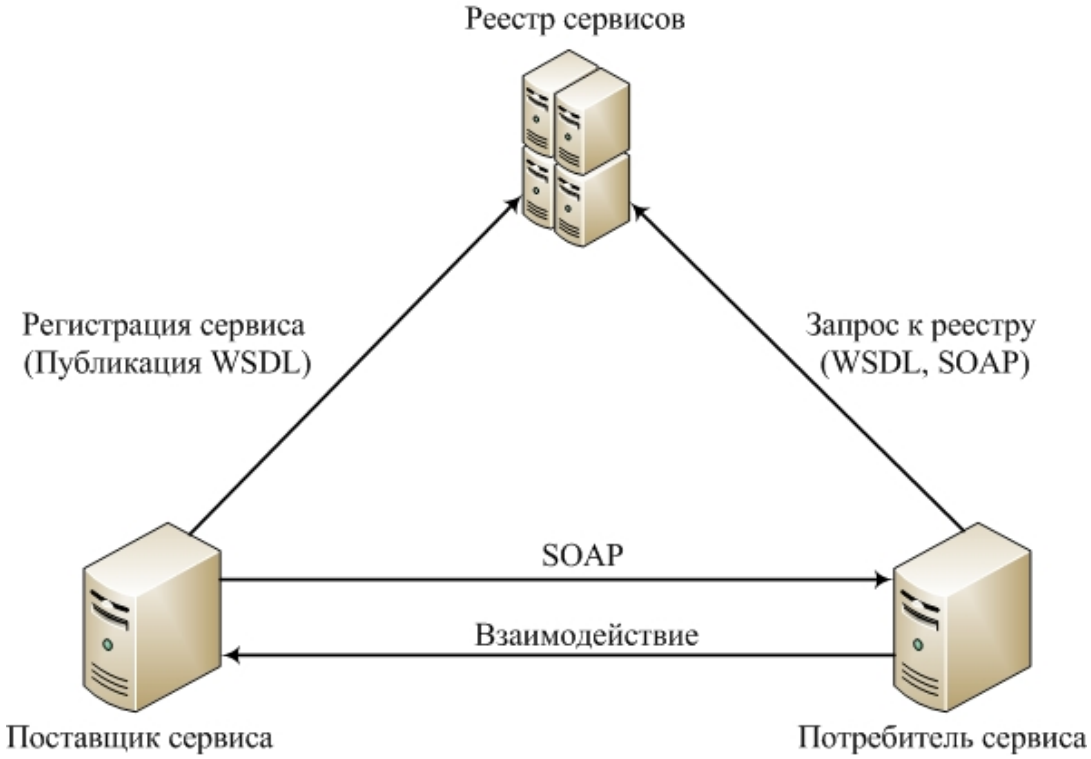


Рис. 1.3. Взаимодействие в рамках сервис-ориентированной архитектуры

В связи с тем, что зачастую, клиент не располагает никакими сведениями о вызываемом сервисе, в реестре хранятся или ссылки на WSDL-описания сервисов, или же сами WSDL-описания.

Важно отметить, что на практике некоторые компоненты, такие как реестр сервисов могут отсутствовать. Например, если число сервисов невелико, и компания больше не расширяется, то, скорее всего не имеет смысла организовывать отдельный реестр для поиска сервисов. Но в случае, если необходимо или собирать некоторую статистику использований сервисов, или анализировать показатели качества обслуживания, то появление дополнительных элементов или даже сложных систем в этой модели взаимодействия будет неизбежно. Использовать связующее программное обеспечение такого рода возможно благодаря применению универсальных контрактов сервисов, а также использованию стандартных протоколов и механизмов взаимодействия.

1.4. Системы управления сервисами

Цель данного программного обеспечения – управление работой сервисов, а также гибкая настройка сервисов относительно бизнес-процесса. Стоит заметить, что

появление этого промежуточного слоя навеяно проблематикой корпоративного взаимодействия. Поэтому начать исследование проблемы необходимо именно с нее.

Для этого рассмотрим следующий пример. Предположим, что есть предприятия А и В. Предприятие А – занимается производством материнских плат, а предприятие В – производством конденсаторов, диодов, резисторов и других комплектующих, которые после производства покупаются предприятием А для производства материнских плат. Оба предприятия давно знают друг друга и взаимодействуют по известным интерфейсам, и в их взаимодействии не возникает никаких проблем. Но вдруг предприятие В в силу природного катаклизма или других причин выбывает из игры и прекращает свою деятельность. Тогда, предприятию А для того, чтобы продолжить бизнес придется решить три существенные задачи. Во-первых, будет необходимо найти подходящих потенциальных партнеров для бизнеса. Во-вторых, каким-то образом необходимо оценить качество предоставляемых ими услуг и выбрать наиболее подходящих партнеров. И в заключение будет необходимо заново реализовать бизнес-процесс с учетом особенностей интерфейсов информационных систем нового партнера.

Вообще, на решение этих проблем может уйти достаточно много времени. Но ведь еще возможны ситуации, когда сотрудничество с новым партнером оказалось неудовлетворительным, и все эти задачи необходимо решать заново. Что касается задач поиска и анализа качества обслуживания, то можно сказать, что эти задачи решаются относительно безболезненно. Ведь искать новых партнеров и собирать сведения о них и их услугах можно и вообще без какой-либо дополнительной корпоративной системы. А вот задача реорганизации бизнес-процесса с учетом новых интерфейсов предполагает, что, скорее всего, будет необходимо или каким-то образом модифицировать типы входных и выходных данных для своих сервисов, или реализовывать новые сервисы для того, чтобы проинтегрировать информационные системы обоих предприятий. И это, даже несмотря на относительно простую реализацию сервисов, может стоить существенных затрат. Особенно, если партнеров менять примерно раз в месяц.

Следовательно, необходимы системы, которые бы позволяли автоматизировать все эти процессы и принимать решение, по возможности, без участия человека.

Конечно, в задачах поиска и, особенно, анализа качества обслуживания такого добиться очень трудно. Но решить проблему различной семантики данных, а также гибкого управления бизнес-процессами не только желательно, но и необходимо.

Можно сказать, что эти три вышеперечисленные проблемы являются основными проблемами корпоративного взаимодействия и решением каждой из них занимается специализированная система.

Таблица 1

Классификация систем управления сервисами

	Проблема	Система управления сервисами, решающая данную проблему
	Поиск наиболее подходящих партнеров	Поисковый реестр
	Оценка качества предлагаемых ими услуг	Система анализа качества обслуживания (QoS)
	Реализация бизнес-процесса, учитывая различные интерфейсы информационных систем предприятий	Сервисная шина предприятия (ESB)

Существуют, как и реализации какой-либо отдельной системы, например системы анализа качества обслуживания, так и реализации, в которых совмещены различные типы систем. Основная проблема в изучении этих систем заключается в том, что почти все из них являются закрытыми и мало документированными, и, как следствие, практически неизвестными. Конечно, есть малое число свободно распространяемых систем, но, зачастую, многие предприятия предпочитают им платные продукты известных производителей. Это связано, во-первых, с более широкими функциональными возможностями и более высоким качеством платных систем, а, во-

вторых, с тем, что производители программного обеспечения подобного рода предлагают техническую поддержку своим покупателям.

1.4.1. Поисковый реестр

Поисковые реестры являются важнейшим звеном сервис-ориентированной архитектуры, и предоставляют клиентам широкие возможности поиска наиболее подходящих для них сервисов. В главе 6 приводится обзор поисковых реестров, а также другая информация, связанная с данными компонентами.

1.4.2. Системы анализа качества обслуживания

Системы анализа QoS – это специализированные системы, анализирующие различные показатели качества обслуживания. В сервис-ориентированной архитектуре можно выделить три основные группы систем анализа качества обслуживания.

Первая группа систем включает в себя системы, которые собирают и анализируют статистику об использованиях сервисов, в частности сведения о производительности, частоте использования, а так же иногда и о характере возвращаемых данных. Такие системы, как правило, служат для определения и устранения “узких мест” в реализациях бизнес-процессов. Как правило, системы подобного рода бывают интегрированы в сервисные шины предприятия, которые будут рассмотрены в следующем разделе.

Вторая группа систем включает в себя системы, которые чаще всего представлены в составе поисковых реестров сервисов. Смысл их работы заключается в следующем. Предположим, что есть некоторый набор тематик, к которым можно отнести зарегистрированный в реестре сервис. Когда поставщик сервиса регистрирует сервис, то он указывает весовые коэффициенты данного сервиса по отношению к каждой из тематик. Причем, чем больше значение коэффициента, тем в большей степени суть сервиса соответствует данной тематике. Когда пользователь осуществляет поиск сервисов в данном реестре, то помимо прочих поисковых

параметров, он указывает предпочтительные весовые коэффициенты для искомого сервиса, и система подбирает наиболее подходящие.

Третья группа систем – это системы сбора пользовательских оценок. Например, после обращения к сервису, в зависимости от времени обслуживания, или характера возвращаемых данных пользователь может проставить оценки данному сервису по одной или нескольким шкалам. Также при поиске сервисов пользователь может указать требуемый уровень оценок. Системы данного типа, тоже, как правило, представлены в составе поисковых реестров.

В принципе, возможны реализации, при которых системы второй и третьей группы не будут реализованы в составе поискового реестра. Но в данном случае, пользователь при поиске сервиса будет обращаться к системе анализа качества обслуживания, которая, в свою очередь, обратится к поисковому реестру, и вернет пользователю лишь те результаты, которые соответствуют его предпочтениям.

При организации как межкорпоративного, так и внутрикорпоративного взаимодействия пользователю должны быть доступны лишь системы второго и третьего типов. Как правило, система первого типа служит для анализа производительности, и ее отчеты представляют ценность для администраторов и разработчиков.

Важно отметить, что наиболее существенной проблемой оценки качества обслуживания в сервис-ориентированной архитектуре является применение весовых коэффициентов, потому что зачастую пользователю не удается должным образом указать требуемые коэффициенты при совершении поискового запроса, или же поставщик сервиса допускает неточности при его регистрации. Из-за этого эффективность поиска значительно падает. Но на сегодняшний день, использование систем весовых коэффициентов являются практически единственным способом задания предпочтений при поиске.

1.4.3. Сервисные шины предприятия

Сервисные шины предприятия (ESB – Enterprise Service Bus) являются самым распространенным видом связующего программного обеспечения. Основная задача

сервисной шины - обеспечение взаимосвязи между различными приложениями по различным протоколам взаимодействия. Таким образом, сервисная шина является интеграционным программным обеспечением, и на предприятиях используется для объединения всех работающих систем, сервисов и приложение в единое информационное пространство. Такой подход использует для упрощения взаимодействия между различными сервисами и приложениями. Также при таком подходе обеспечивается гибкость и масштабируемость. Например, при замене одного приложения, подключенного к шине, не нужно вносить изменения в другие приложения или их конфигурацию. Типичная схема сервисной шины предприятия приведена далее.

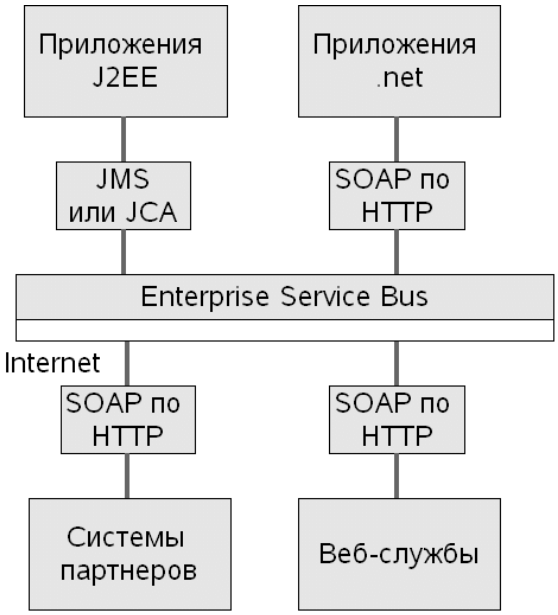


Рис. 1.5. Сервисная шина предприятия

Одно из важнейших преимуществ при использовании ESB заключается в том, что, например, если при реализации бизнес-процесса необходимо последовательно обратиться к нескольким сервисам, и каким-либо образом модифицированные выходные данные одного из них будут являться входными данными для следующего сервиса, то в программном коде не нужно будет производить эту модификацию выходных данных. Правила преобразования данных можно задавать непосредственно через утилиту конфигурирования самой шины.

Среди основных возможностей, которые предоставляет сервисная шина предприятия, можно выделить следующие.

Возможность задавать бизнес-процессы;

Возможность задавать бизнес-правила;

Возможность отслеживать выполнение бизнес-процессов;

Возможность подключать новые сервисы, унаследованные приложения и другие источники данных;

Возможность определять преобразования данных между различными источниками, включая сервисы;

Возможность осуществлять централизованное управление и внесение изменений.

Таким образом, использование сервисной шины освобождает разработчика от реализации интеграционной логики, а также логики преобразования данных от разных источников.

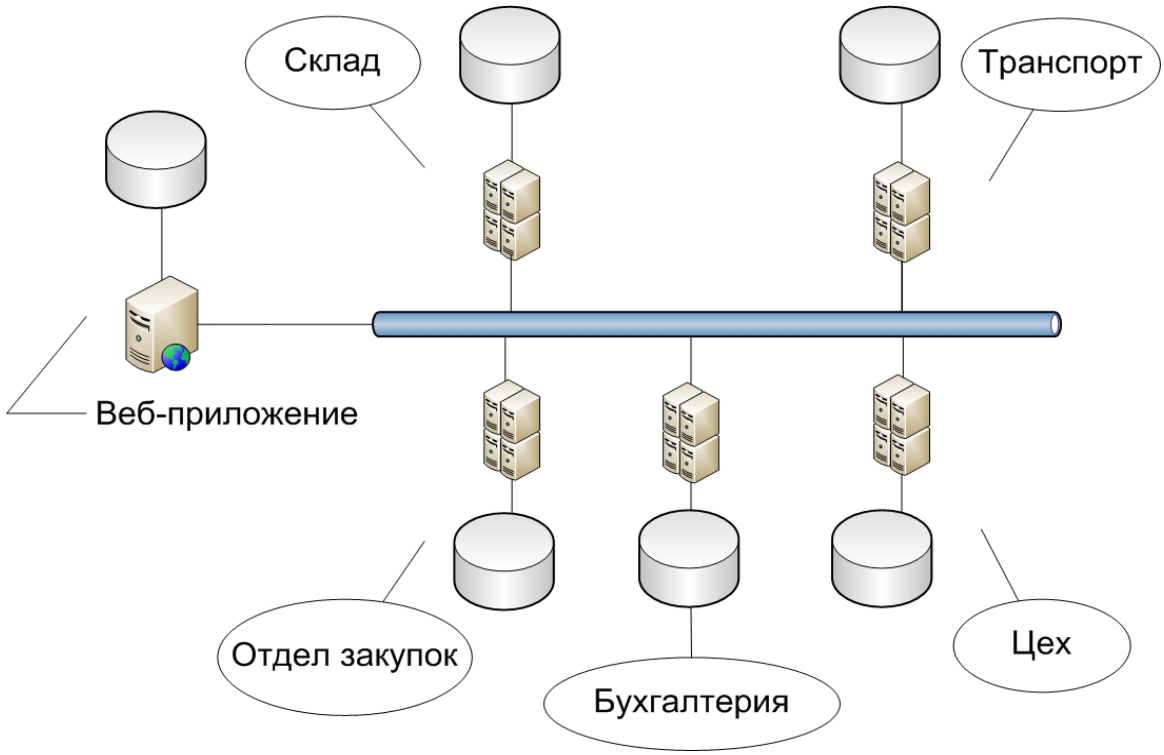


Рис. 1.6. Интеграция с помощью сервисной шины

К тому же современные сервисные шины имеют возможности работы с различными источниками данных – базами данных, файлами. Следовательно, можно не разрабатывать отдельные сервисы, которые, осуществляли только работу с источником данных. Данную задачу можно выполнить только с участием сервисной шины. Это позволяет уменьшить сроки разработки.

При использовании сервисной шины значительно уменьшается время модификации системы, а также увеличивается фактор повторного использования. Время модификации удается снизить благодаря вынесению все интеграционной логики на уровень шины данных. А фактор повторного использования увеличивается благодаря тому, что один и тот же сервис может участвовать сразу же в нескольких бизнес-процессах.

Важно отметить, что современные сервисные шины имеют очень развитые и простые в использовании консоли администрирования, через которые можно создавать и модифицировать бизнес-процессы. Благодаря этому интеграцию не обязательно проводить разработчику или администратору системы, а эту задачу может выполнить, например, бизнес-аналитик.

Использование сервисной шины принесет пользу, как и при обеспечении межкорпоративного взаимодействия, так и при реализации внутрикорпоративного. В любом случае обеспечивается минимальное время внесения изменений, и, как следствие, наибольшая скорость реакции на изменяющиеся условия внешней среды.

Наиболее известными сервисными шинами являются WebSphere от компании IBM, webMethods компании Software AG, BizTalk от Microsoft, JBoss ESB, поддерживаемая компанией RedHat, Mule ESB, WSMX. Причем последние три продукта являются свободно-распространяемыми.

1.4.4. Обзор системы управления сервисами WSMX

WSMX (Web Service Modeling eXecution environment) – наиболее типичная система управления сервисами. Содержит в себе ESB, поисковый реестр, и средство управления качеством обслуживания.

WSMX служит для динамического обнаружения, выбора и вызова сервисов. Ядром WSMX является менеджер WSMX. Он состоит из нескольких основных компонентов: менеджер ресурсов, поисковик, выборщик и посредник.

Для того чтобы веб-сервис был доступен через эту систему его необходимо зарегистрировать в ней. Для этого менеджер, администратор или поставщик сервиса используя веб-интерфейс системы и язык моделирования веб-сервисов WSML

описывают тематики сервиса, бизнес-процессы в которых он участвует, а так же необходимые преобразования входных и выходных данных в соответствии с логикой бизнес-процесса.

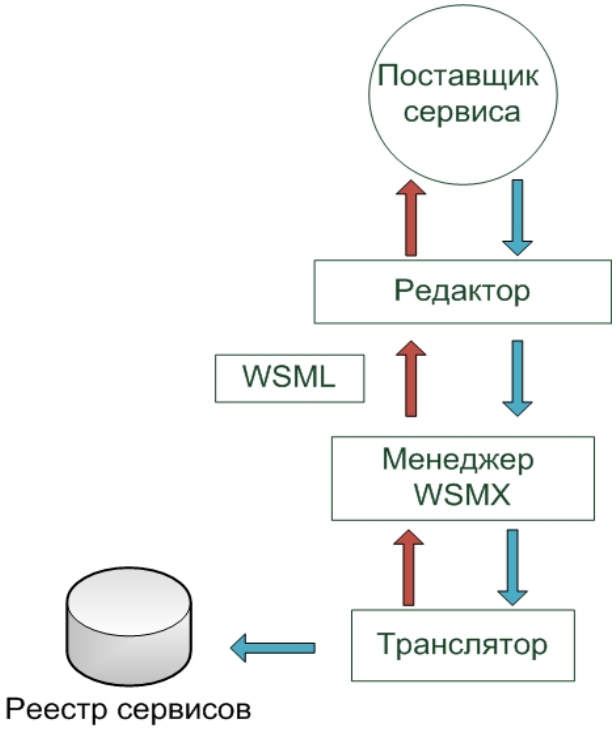


Рис. 1.7. Механизм регистрации сервиса в системе WSMX

Затем менеджер WSMX, используя специальный транслятор преобразует абстрактные описания сервиса в специальные метаданные, которые записываются в реестре сервисов. Эти данные в дальнейшем будут использоваться при обнаружении сервиса и его вызове.

Выборщик WSMX помогает выбрать лучший сервис из возможного набора подходящих сервисов. При выборе учитываются различные критерии пользователей.

В WSMX для организации взаимодействия применяются посредники. WSMX имеет два типа посредников: посредник данных и посредник процессов. Посредник данных используется для преодоления семантических различий между информацией из различных источников. Посредник процессов позволяет избавиться от несовпадений в протоколах взаимодействия между провайдером и потребителем сервиса. Это бывает актуально, когда, например, сервис и клиент используют разные версии протокола SOAP.

Поиск сервиса в WSMX осуществляется в соответствии со схемой, изображенной на рисунке ниже.

Рис. 1.8. Механизм поиска сервиса в системе WSMX

Потребитель сервиса отправляет поисковый запрос. Данный запрос анализируется, и с помощью менеджера WSMX и особого транслятора, данные поискового запроса трансформируются в набор метаданных. Далее по этим метаданным осуществляется поиск сервисов. В последнюю очередь, результаты поиск фильтруются выборщиком в соответствии с заданными критериями пользователя. После этого результат поиска возвращается потребителю.

WSMX также предоставляет интерфейс для приема пользовательских запросов. Как правило, приходящий пользовательский запрос несет в себе идентификатор требуемого бизнес-процесса и набор входных данных. Данные в зависимости от логики бизнес-процесса преобразуются к требуемому виду.

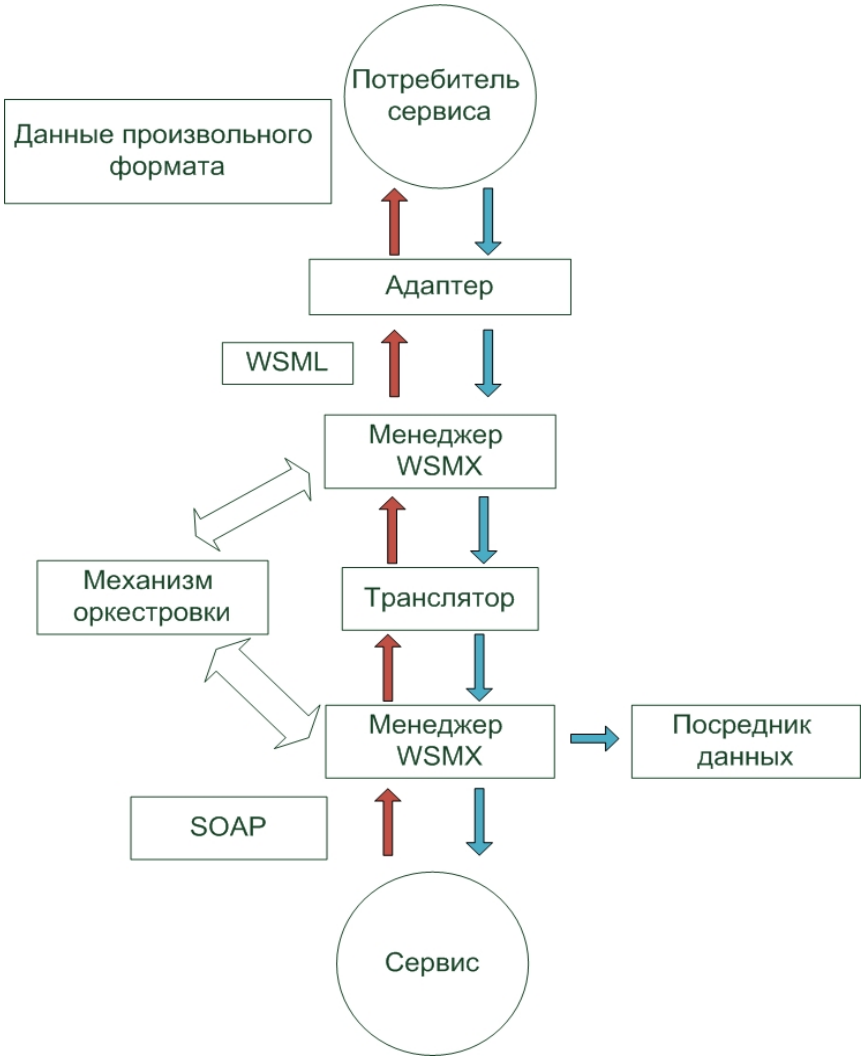


Рис. 1.9. Механизм выполнения бизнес-процесса в системе WSMX

После чего, менеджер WSMX, при помощи заложенных в него механизмов оркестровки последовательно обращается к одному или нескольким сервисам, используя протокол SOAP.

Если помимо вызова сервисов необходимо получить данные из какого-либо источника, например базы данных, то эту задачу выполняет посредник данных.

Последняя стадия данного взаимодействия – возврат результата запроса потребителю. Здесь происходит процесс преобразования данных к требуемому виду, после чего результат возвращается клиенту.

Глава 2. Технологии разработки SOAP веб-сервисов

2.1. Процесс разработки

Для того чтобы разработать веб-сервис не достаточно верно реализовать бизнес-логику. Если абстрагироваться от платформы разработки и ее инструментов, то в общем случае процесс разработки веб-сервиса состоит из следующих шагов:

1. Определение интерфейса веб-сервиса. На данном шаге необходимо определить, какие операции предоставляет веб-сервис, а также их входные и выходные данные.
2. Реализация этого интерфейса.
3. Создание WSDL-описания веб-сервиса.
4. Разработка сетевого модуля-обработчика запросов-ответов. Цель данного модуля – преобразование входного сообщения в формате XML в объекты тех классов, с которыми оперирует веб-сервис, а также маршаллинг объектов, которые являются выходными данными сервиса в формат XML.
5. Запуск веб-сервиса. После запуска сервис должен корректно обрабатывать запросы.
6. Публикация WSDL-описания.

Вполне очевидно, что среди перечисленных пунктов только первые два имеют непосредственное отношение к бизнес-логике и функциональности. Остальные же являются некими “накладными расходами”.

Также понятно, что описанные в пунктах 3-6 процессы абсолютно одинаковы для всех веб-сервисов, следовательно, их можно автоматизировать. Поэтому один из важнейших критериев при выборе платформы для разработки веб-сервисов – это возможности автоматической генерации WSDL-описания по интерфейсу сервиса, а генерация интерфейса по WSDL-описанию, его автоматическая публикация, поддержка сетевого взаимодействия без написания дополнительного программного кода.

В данном пособии подробно рассматривается процесс разработки веб-сервисов на платформе Java. Все приведенные примеры были разработаны с использованием

JDK 7u45, IDE NetBeans 7.4. В качестве сервера приложений использован GlassFish 4.0, а СУБД – PostgreSQL 9.3.1. Для тестирования веб-сервисов использовано приложение SoapUI 4.6.

2.2. Обзор JAX-WS

Для разработки веб-сервисов SOAP в Java основным стандартом является JAX-WS. JAX-WS предоставляет разработчику множество возможностей гибкой настройки сервисов с помощью аннотаций языка Java. При его использовании взаимодействие клиента с сервисом полностью скрыто от разработчика, что дает ему возможность как можно больше сконцентрироваться на реализации бизнес-логики. А возможности автоматической генерации и публикации WSDL и XSD, а также кода клиента веб-сервиса значительно ускоряют процесс разработки.

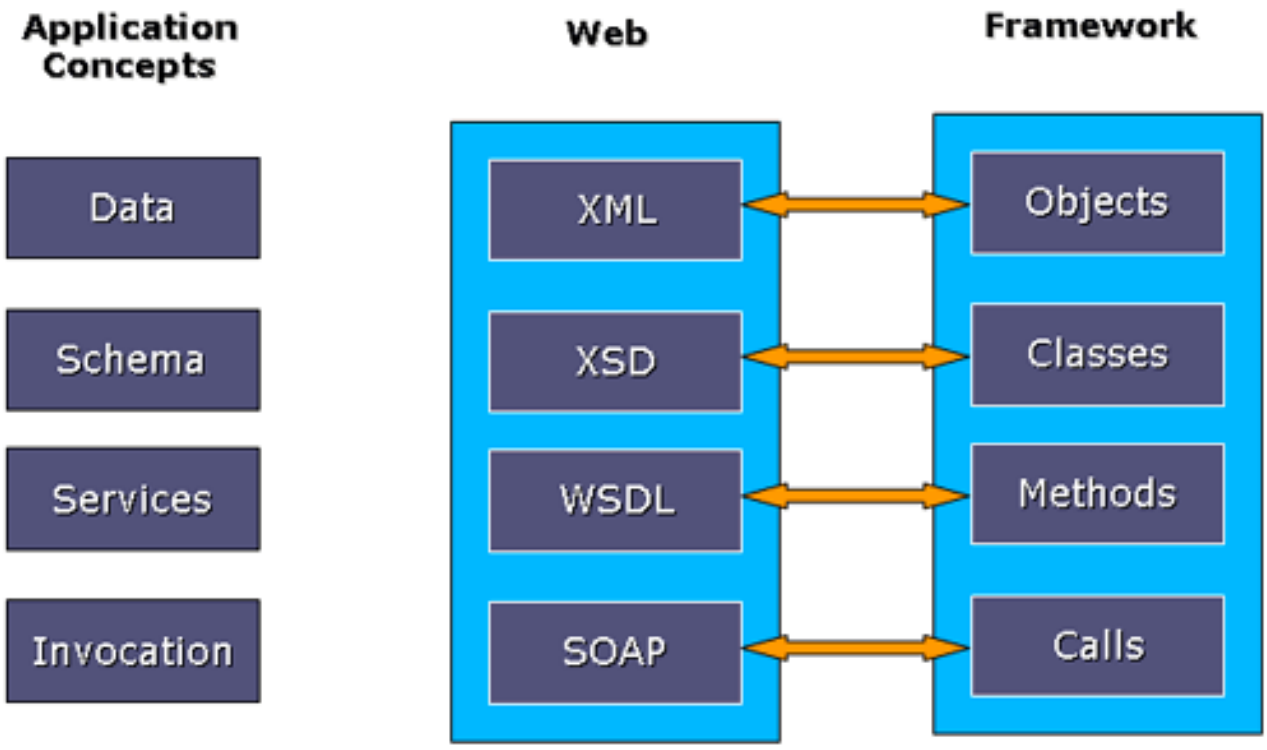


Рис. 2.0. Соотношение сущностей программного кода и SOAP-сервисов

При использовании JAX-WS работа с удаленным сервисом ведется, будто с локальным объектом через специальный интерфейс. Методы этого интерфейса соответствуют операциям веб-сервиса, описанным в WSDL. Классы, которые участвуют во взаимодействии клиента и сервиса описаны в XSD-схеме,

опубликованной вместе с WSDL этого сервиса. А передаваемые объекты преобразуются в XML.

2.2.1. wsgen

Утилита wsgen используется для генерации WSDL и XSD. Важно отметить, что wsgen производит анализ классов веб-сервиса, а затем на основании анализа продуцирует описание сервиса. Это очень важно, поскольку программисту необходимо только написать код классов для веб-сервиса.

Пример использования: wsgen com.service.OrderProcessService -wsdl

В данном случае будет сгенерировано WSDL-описание веб-сервиса на основании класса OrderProcessingService. Следует отметить, что для корректной работы wsgen в classpath должны находиться все классы, которые используются в интерфейсе, класса OrderProcessingService.

2.2.2. wsimport

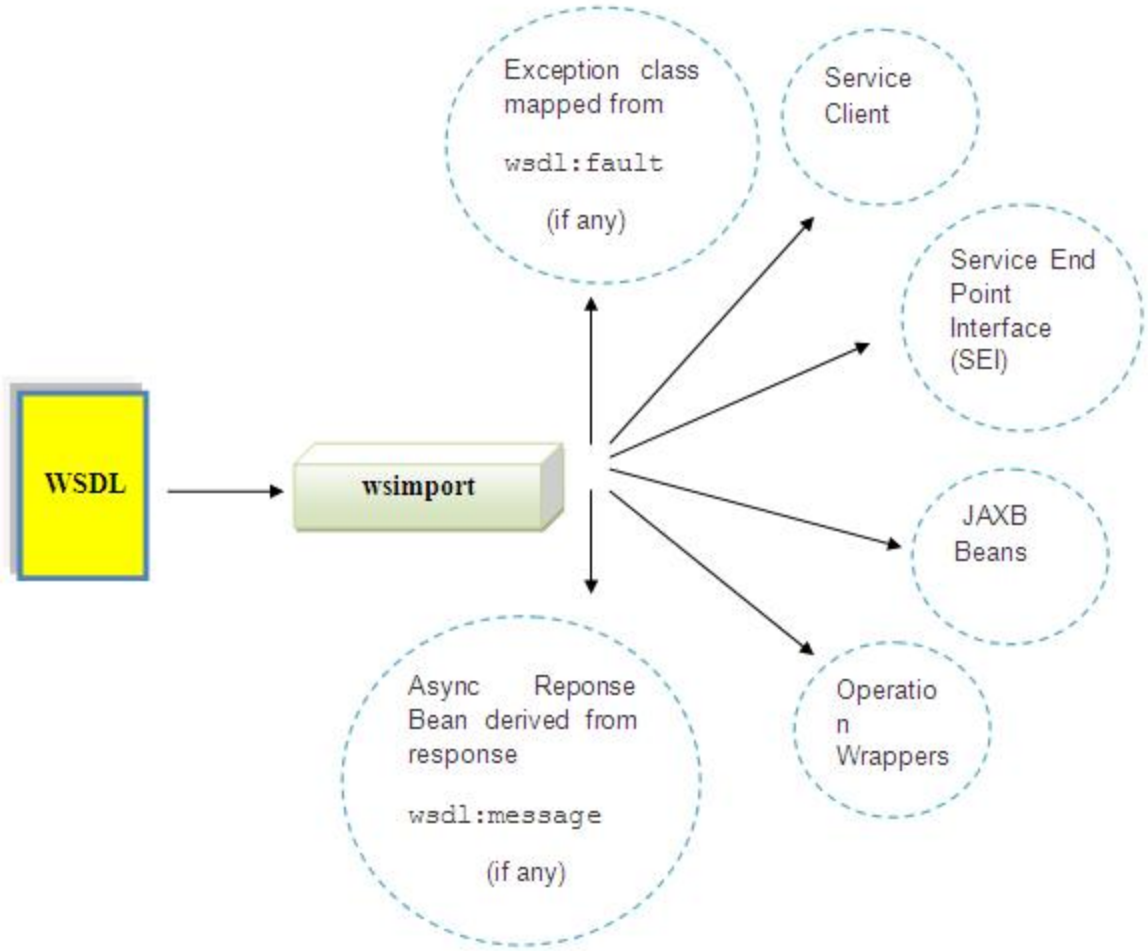


Рис. 2.1. wsimport

Рассмотренная выше утилита wsgen осуществляет генерацию описаний по классам веб-сервиса. Утилита wsimport осуществляет обратную операцию, а именно генерирует Java-код, необходимый для взаимодействия с веб-сервисом.

Таким образом, из исходного WSDL-документа генерируются:

- классы исключений, которые могут возникнуть при взаимодействии с сервисом
- реализация клиента для сервиса
- интерфейс сервиса
- объекты задействованные в интерфейсе веб-сервиса
- средства для асинхронного обращения к веб-сервису (в случае, если таковые предусмотрены)

Пример использования: wsimport –p stockquote http://stockquote.xyz/quote?wsdl

В результате такого запуска wsimport будут сгенерированы вышеописанные сущности на основании описания, расположенного по адресу http://stockquote.xyz/quote?wsdl. Также сгенерированные артефакты будут находиться в пакете stockquote.

2.3. Реализация веб-сервиса

2.3.1. Подготовка

В данном разделе будет рассмотрена реализация веб-сервиса, а также клиента к нему и будет проведено его тестирование. Разрабатываемый веб-сервис должен осуществлять выборку из таблицы Persons базы данных и возвращать результат этой выборки.

В данном разделе будет показано две реализации этого сервиса. Первая из них представляет собой standalone приложение, вторая же – Java веб-приложение, предназначенное для развертывания на сервере приложений.

Для создания таблицы в базе данных следует использовать следующий SQL:

```
CREATE TABLE "persons" (
  id bigserial NOT NULL,
  name character varying(200),
  surname character varying(200),
  age integer,
  CONSTRAINT "Persons_pkey" PRIMARY KEY (id)
);
```

Для добавления записей в таблицу следует выполнить:

```
INSERT INTO persons(name, surname, age) values ('Петр', 'Петров', 25);
INSERT INTO persons(name, surname, age) values ('Владимир', 'Иванов', 26);
INSERT INTO persons(name, surname, age) values ('Иван', 'Иванов', 27);
INSERT INTO persons(name, surname, age) values ('Иммануил', 'Кант', 28);
INSERT INTO persons(name, surname, age) values ('Джордж', 'Клуни', 29);
INSERT INTO persons(name, surname, age) values ('Билл', 'Рубцов', 30);
INSERT INTO persons(name, surname, age) values ('Марк', 'Марков', 31);
INSERT INTO persons(name, surname, age) values ('Галина', 'Матвеева', 32);
INSERT INTO persons(name, surname, age) values ('Святослав', 'Павлов', 33);
```

```
INSERT INTO persons(name, surname, age) values ('Ольга', 'Бергольц', 34);
INSERT INTO persons(name, surname, age) values ('Лев', 'Рабинович', 35);
```

2.3.2. Реализация сервиса

Основные классы сервиса следующие:

Класс	Ответственность
App	Содержит main-метод, осуществляет запуск сервиса
ConnectionUtil	Утилитный класс для создания подключений к базе данных
Person	POJO, соответствует сущности, описанной в таблице persons базы данных
PersonWebService	Веб-сервис. Содержит операции веб-сервиса
PostgreSQLDAO	Data Access Object

Далее приведен исходный код классов и некоторые комментарии.

App.java

```
package com.wishmaster.ifmo.ws.jaxws;

import javax.xml.ws.Endpoint;

public class App {

    public static void main(String[] args) {
        String url = "http://0.0.0.0:8080/PersonService";
        Endpoint.publish(url, new PersonWebService());
    }
}
```

Данный класс содержит main метод, и его основная цель – это запустить веб-сервис. Важно отметить, что такой код нужен только для запуска сервиса как standalone-приложение. При реализации сервиса в качестве приложения, функционирующего в составе сервера приложений, запуск сервиса осуществляется непосредственно сервером приложений.

Для запуска веб-сервисов используется класс Endpoint, со статическими методами publish. Данный метод принимает на вход URL, по которому сервис будет доступен, а также класс-реализацию веб-сервиса.

ConnectionUtil.java

```
package com.wishmaster.ifmo.ws.jaxws;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ConnectionUtil {

   private static final String JDBC_URL = "jdbc:postgresql://localhost:5432/ifmo-ws";
   private static final String JDBC_USER = "ifmo-ws";
   private static final String JDBC_PASSWORD = "ifmo-ws";

   static {
      try {
         Class.forName("org.postgresql.Driver");
      } catch (ClassNotFoundException ex) {
         Logger.getLogger(PostgreSQLDAO.class.getName()).log(Level.SEVERE, null, ex);
      }
   }

   public static Connection getConnection() {
      Connection connection = null;
      try {
         connection = DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD);
      } catch (SQLException ex) {
         Logger.getLogger(ConnectionUtil.class.getName()).log(Level.SEVERE, null, ex);
      }
      return connection;
   }
}
```

Данный утилитный класс используется для получения JDBC-соединений с базой данных. Обратите внимание на то, что параметры соединения заданы в статических переменных класса. Вам необходимо задать в них значения для соединения с Вашей СУБД.

Person.java

```
package com.wishmaster.ifmo.ws.jaxws;

public class Person {
   private String name;
   private String surname;
   private int age;

   public Person() {
   }

   public Person(String name, String surname, int age) {
      this.name = name;
      this.surname = surname;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", surname=" + surname + ", age=" + age + '}';
    }
}
```

Объекты этого класса возвращает веб-сервис. Обратите внимание на соответствие класса JavaBeans-стандарту: наличие конструктора без параметров, а также get/set-методы для private-полей.

PersonWebService.java

```
package com.wishmaster.ifmo.ws.jaxws;

import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(serviceName = "PersonService")
public class PersonWebService {

    @WebMethod(operationName = "getPersons")
    public List<Person> getPersons() {
        PostgreSQLDAO dao = new PostgreSQLDAO();
```

```
        List<Person> persons = dao.getPersons();
        return persons;
    }

}
```

Класс-реализация веб-сервиса. Данный сервис предоставляет одну операцию – getPersons. Обратите внимание на аннотации @WebService и @WebMethod. @WebService используется для того, чтобы маркировать класс, который должен предоставлять функциональность через веб-сервис. @WebMethod – маркер метода, который будет являться операцией веб-сервиса.

Заметьте, что метод getPersons() возвращает List<Person>, а не какой-либо специальный объект. То есть, единственное, что нужно указать разработчику – это аннотация над методом, а об остальном позаботится имплементация JAX-WS.

PostgreSQLDAO.java

```
package com.wishmaster.ifmo.ws.jaxws;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class PostgreSQLDAO {

    public List<Person> getPersons() {
        List<Person> persons = new ArrayList<>();
        try (Connection connection = ConnectionUtil.getConnection()){
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("select * from persons");

            while (rs.next()) {
                String name = rs.getString("name");
                String surname = rs.getString("surname");
                int age = rs.getInt("age");

                Person person = new Person(name, surname, age);
                persons.add(person);
            }
        } catch (SQLException ex) {
            Logger.getLogger(PostgreSQLDAO.class.getName()).log(Level.SEVERE, null, ex);
        }

        return persons;
    }
}
```

Класс, содержащий метод для выборки данных из базы данных, а также упаковки этих данных в объекты класса Person.

Для успешной работы в classpath необходимо иметь классы JDBC-драйвера PostgreSQL. При сборке с помощью maven добавьте следующую зависимость:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.2-1003-jdbc4</version>
</dependency>
```

2.4. Запуск сервиса и просмотр его описаний

После компиляции и запуска, по адресу http://localhost:8080/PersonService?wsdl будет доступно WSDL-описание сервиса, которое сгенерировано автоматически. Именно это и подразумевалось под автоматической генерацией описаний веб-сервиса.

Далее приведено WSDL-описание данного сервиса:

```
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://jaxws.ws.ifmo.wishmaster.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://jaxws.ws.ifmo.wishmaster.com/" name="PersonService">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://jaxws.ws.ifmo.wishmaster.com/" schemaLocation="http://localhost:8080/PersonService?xsd=1"/>
        </xsd:schema>
    </types>
    <message name="getPersons">
        <part name="parameters" element="tns:getPersons"/>
    </message>
    <message name="getPersonsResponse">
        <part name="parameters" element="tns:getPersonsResponse"/>
    </message>
    <portType name="PersonWebService">
        <operation name="getPersons">
            <input message="tns:getPersons"/>
            <output message="tns:getPersonsResponse"/>
        </operation>
    </portType>
    <binding name="PersonWebServicePortBinding" type="tns:PersonWebService">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="getPersons">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
```

```
            <soap:body use="literal"/>
          </output>
        </operation>
      </binding>
      <service name="PersonService">
        <port name="PersonWebServicePort" binding="tns:PersonWebServicePortBinding">
          <soap:address location="http://localhost:8080/PersonService"/>
        </port>
      </service>
    </definitions>
```

В данной WSDL видно, что веб-сервис имеет одну операцию getPersons, а также то, что XSD-схема расположена по URL: http://localhost:8080/PersonService?xsd=1. Содержимое этой XSD-схемы представлено далее:

```
    <xs:schema xmlns:tns="http://jaxws.ws.ifmo.wishmaster.com/"
xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
targetNamespace="http://jaxws.ws.ifmo.wishmaster.com/">
      <xs:element name="getPersons" type="tns:getPersons"/>
      <xs:element name="getPersonsResponse" type="tns:getPersonsResponse"/>
      <xs:complexType name="getPersons">
        <xs:sequence/>
      </xs:complexType>
      <xs:complexType name="getPersonsResponse">
        <xs:sequence>
          <xs:element name="return" type="tns:person" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="person">
        <xs:sequence>
          <xs:element name="age" type="xs:int"/>
          <xs:element name="name" type="xs:string" minOccurs="0"/>
          <xs:element name="surname" type="xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
```

В данной XSD-схеме присутствует описание типа person, который по составу полей эквивалентен классу Person.

2.5. Тестирование сервиса с помощью SoapUI

Для того, чтобы обращаться к нашему сервису через SoapUI, необходимо в SoapUI создать проект (File -> New SOAP Project) и в появившемся окне указать имя этого проекта, а также адрес WSDL тестируемого сервиса. После создания проекта станет возможным отправка запросов и просмотр ответов от веб-сервиса.

Теперь протестируем созданный сервис. Метод getPersons не принимает на вход никаких параметров, поэтому оставим исходный, сгенерированный XML в SoapUI без изменений и выполним запрос.

SOAP-запрос:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:jax="http://jaxws.ws.ifmo.wishmaster.com/">
  <soapenv:Header/>
  <soapenv:Body>
    <jax:getPersons/>
  </soapenv:Body>
</soapenv:Envelope>
```

Из этого запроса видно, что вызывается операция getPersons.

Если подготовительные работы по созданию таблицы и импорту данных выполнены, параметры подключения заданы верно и веб-сервис запущен без ошибок, то можно будет увидеть (справа от запроса) следующий SOAP-ответ:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getPersonsResponse xmlns:ns2="http://jaxws.ws.ifmo.wishmaster.com/">
      <return>
        <age>25</age>
        <name>Петр</name>
        <surname>Петров</surname>
      </return>
      <return>
        <age>26</age>
        <name>Владимир</name>
        <surname>Иванов</surname>
      </return>
      <return>
        <age>27</age>
        <name>Иван</name>
        <surname>Иванов</surname>
      </return>
      <return>
        <age>28</age>
        <name>Иммануил</name>
        <surname>Кант</surname>
      </return>
      <return>
        <age>29</age>
        <name>Джордж</name>
        <surname>Клуни</surname>
      </return>
      <return>
        <age>30</age>
        <name>Билл</name>
        <surname>Рубцов</surname>
      </return>
      <return>
        <age>31</age>
        <name>Марк</name>
        <surname>Марков</surname>
      </return>
      <return>
```

```
            <age>32</age>
            <name>Галина</name>
            <surname>Матвеева</surname>
          </return>
          <return>
            <age>33</age>
            <name>Святослав</name>
            <surname>Павлов</surname>
          </return>
          <return>
            <age>34</age>
            <name>Ольга</name>
            <surname>Бергольц</surname>
          </return>
          <return>
            <age>35</age>
            <name>Лев</name>
            <surname>Рабинович</surname>
          </return>
        </ns2:getPersonsResponse>
      </S:Body>
    </S:Envelope>
```

По сути, в этом ответе можно увидеть записи таблицы persons, представленные в формате XML. Именно так выглядят запросы и ответы протокола SOAP. Именно в таком виде они отправляются клиентом и сервисом.

2.6. Разработка клиента веб-сервиса

В предыдущем разделе было выполнено тестирование веб-сервиса и в роли клиента веб-сервиса выступала утилита SoapUI. Теперь необходимо разработать клиент для веб-сервиса самостоятельно для того, чтобы глубже понять процесс взаимодействия между сервисом и клиентом JAX-WS.

При разработке клиента сначала необходимо сгенерировать необходимые классы из исходного WSDL-описания сервиса. Для этого следует воспользоваться ранее описанной утилитой wsimport. В современных IDE есть графическая поддержка wsimport, и благодаря этому, импорт можно удобно осуществить без ввода параметров в командную строку.

Далее на скриншоте показана форма генерации кода на основе WSDL-описания сервиса.

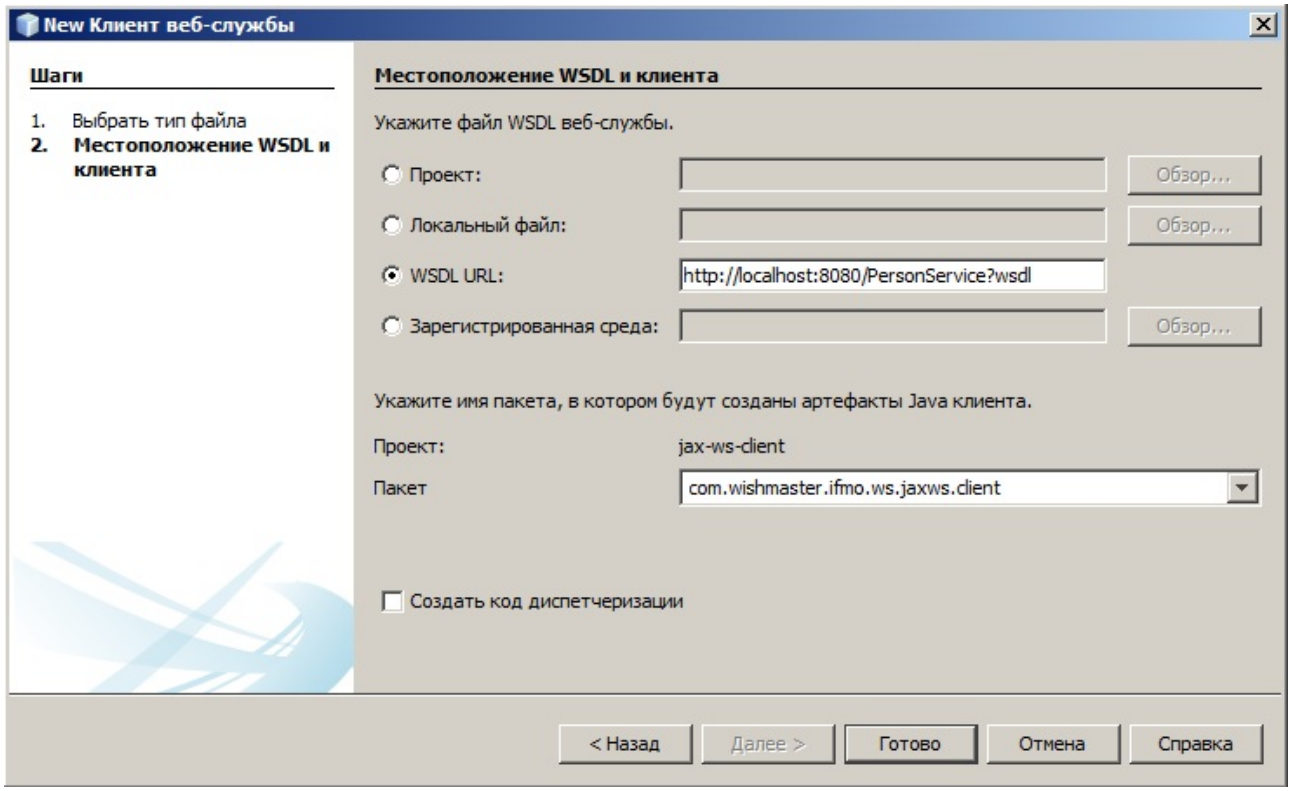


Рис. 2.2. Использование wsimport в NetBeans IDE

После выполнения утилиты wsimport можно воспользоваться сгенерированными классами для обращения к веб-сервису. В приложении-клиенте веб-сервиса будет только один класс.

Класс	Ответственность
WebServiceClient	Содержит main-метод, использует сгенерированные классы для обращения к веб-сервису

WebServiceClient.java

```
package com.wishmaster.ifmo.ws.jaxws.client;

import com.wishmaster.ifmo.ws.jaxws.client.generated.Person;
import com.wishmaster.ifmo.ws.jaxws.client.generated.PersonService;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;

public class WebServiceClient {
```

```
    public static void main(String[] args) throws MalformedURLException {
        URL url = new URL("http://localhost:8080/PersonService?wsdl");
        PersonService personService = new PersonService(url);
        List<Person> persons = personService.getPersonWebServicePort().getPersons();
        for (Person person : persons) {
            System.out.println("name: " + person.getName() +
                    ", surname: " + person.getSurname() + ", age: " + person.getAge());
        }
        System.out.println("Total persons: " + persons.size());
    }
}
```

Для того чтобы осуществить обращение к веб-сервису, необходимо создать экземпляр класса клиента веб-сервиса (PersonService), в конструктор которого передается URL, где расположено WSDL-описание. Затем необходимо из этого клиента получить SOAP-port веб-службы (getPersonWebServicePort), и после уже вызвать требуемый метод веб-сервиса (getPersons). Класс Person, который сгенерирован, имеет поля name, surname, age. Этот набор полей эквивалентен набору полей класса Person, который был создан при реализации веб-сервиса. Следует обратить внимание, насколько элегантно происходит взаимодействие с удаленным сервисом. Таким образом, сетевое взаимодействие полностью скрыто от программиста – ему достаточно только на локальном объекте вызвать метод веб-сервиса в то время, как все остальное будет сделано силами JAX-WS-имплементации.

После выполнения этого кода на консоль будут выведены те же "персоны", которые были ранее добавлены в базу данных.

Не забудьте, что веб-сервис должен быть запущен при запуске кода клиента.

Приведенный выше пример показывает, насколько просто осуществляется взаимодействие сторон. При использовании такого подхода разработчик полностью огражден от деталей взаимодействия, особенностей структуры трансферных объектов. Также важен тот факт, что разработчику требуется меньше времени для того, чтобы разобраться с трудным API, поскольку данный процесс теперь лишь сводится к

изучению интерфейса веб-сервиса, и вовсе не обязательно читать громоздкие документации.

2.7. Разработка J2EE веб-сервиса

Ранее была продемонстрирована реализация standalone-application веб-сервиса. Для того чтобы показать универсальность JAX-WS, в данном разделе будет показано, как разработать веб-сервис для развертывания его на сервере приложений J2EE. В качестве сервера приложений будет использован GlassFish 4.0.

Основное различие в такой имплементации сервиса заключается, во-первых, в том, что не нужен класс с main-методом, который запускает веб-сервис, и, во-вторых, в том, что можно использовать встроенную в сервер приложений функциональность.

Таким образом, количество строк кода сокращается, по сравнению с реализацией в виде standalone-приложения, но необходимо понимать, что для запуска сервиса на сервере приложений потребуются бОльшие ресурсы, в частности, больше памяти.

Основные классы реализации:

Класс	Ответственность
Person	POJO, соответствует сущности, описанной в таблице persons базы данных
PersonWebService	Веб-сервис. Содержит операции веб-сервиса. Содержит инъекцию источника данных, настроенного на стороне сервера приложений
PostgreSQLDAO	Data Access Object

PersonWebService.java

```
package com.wishmaster.ifmo.ws.jaxws;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.sql.DataSource;
```

```
@WebService(serviceName = "PersonService")
public class PersonWebService {

    @Resource(lookup = "jdbc/ifmo-ws")
    private DataSource dataSource;

    @WebMethod(operationName = "getAllPersons")
    public List<Person> getAllPersons() {
        PostgreSQLDAO dao = new PostgreSQLDAO(getConnection());
        return dao.getAllPersons();
    }

    @WebMethod(operationName = "getPersonsByName")
    public List<Person> getPersonsByName(@WebParam(name = "personName") String name) {
        PostgreSQLDAO dao = new PostgreSQLDAO(getConnection());
        return dao.getPersonsByName(name);
    }

    private Connection getConnection() {
        Connection result = null;
        try {
            result = dataSource.getConnection();
        } catch (SQLException ex) {
            Logger.getLogger(PersonWebService.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

Видно, что реализация основного класса веб-сервиса изменилась незначительно. Класс по-прежнему имеет аннотацию @WebService, а методы аннотинованы, @WebMethod. Единственное различие в том, что в класс добавлено поле DataSource dataSource. Данное поле представляет собой источник данных, в качестве которого выступает база данных PostgreSQL. Ранее мы регистрировали JDBC-драйвер, создавали подключение вручную, а теперь же ответственность за создание подключений возложена на сервер приложений. Для этого на стороне сервера приложений были указаны настройки для соединения с БД, которые ранее были указаны в классе ConnectionUtil, и теперь, вызов метода dataSource.getConnection() возвращает аналогичное соединение. Аннотация над полем dataSource @Resource(lookup = "jdbc/ifmo-ws") означает, что сервер приложений, каждый раз при вызове операции веб-сервиса осуществляет инъекцию ресурса с JNDI-именем "jdbc/ifmo-ws" в поле dataSource. То есть, вручную мы не устанавливаем его значение, а за нас это осуществляет сервер приложений. Такая техника называется Dependency Injection (внедрение зависимостей).

Остальные классы, по сравнению с предыдущей реализацией не изменились. Строго говоря, и класс PersonWebService можно было не менять, а оставить как есть. Данное изменение сделано, чтобы показать преимущества сервера приложений.

В результате сборки проекта данного веб-сервиса будет получен war-архив, представляющий собой веб-приложение, готовое для развертывания на сервере приложений. И если после развертывания этого архива осуществить вызов

веб-сервиса, можно убедиться, в том, что функциональность не изменилась, и поведение сервиса осталось прежним. Например, вызов метода getAllPersons продуцирует такое сообщение:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getAllPersonsResponse xmlns:ns2="http://jaxws.ws.ifmo.wishmaster.com/">
      <return>
        <age>25</age>
        <name>Петр</name>
        <surname>Петров</surname>
      </return>
      <return>
        <age>26</age>
        <name>Владимир</name>
        <surname>Иванов</surname>
      </return>
      <return>
        <age>27</age>
        <name>Иван</name>
        <surname>Иванов</surname>
      </return>
      <return>
        <age>28</age>
        <name>Иммануил</name>
        <surname>Кант</surname>
      </return>
      <return>
        <age>29</age>
        <name>Джордж</name>
        <surname>Клуни</surname>
      </return>
      <return>
        <age>30</age>
        <name>Билл</name>
        <surname>Рубцов</surname>
      </return>
      <return>
        <age>31</age>
        <name>Марк</name>
        <surname>Марков</surname>
      </return>
      <return>
        <age>32</age>
        <name>Галина</name>
        <surname>Матвеева</surname>
      </return>
      <return>
        <age>33</age>
        <name>Святослав</name>
        <surname>Павлов</surname>
      </return>
      <return>
        <age>34</age>
```

```
        <name>Ольга</name>
        <surname>Бергольц</surname>
      </return>
      <return>
        <age>35</age>
        <name>Лев</name>
        <surname>Рабинович</surname>
      </return>
    </ns2:getAllPersonsResponse>
  </S:Body>
</S:Envelope>
```

Сравнив сообщения двух различных реализаций сервиса видно, что их методы возвращаю один и те же данные.

Таким образом, можно подчеркнуть, что универсальность – одно из основных преимуществ технологии веб-сервисов.

2.8. Обработка ошибок

Помимо основных функциональных возможностей, система должна уметь распознавать исключительные ситуации, а также корректно обрабатывать возникающие в процессе работы ошибки.

Для этого в сообщениях SOAP-протокола имеется специальный блок soap:Fault, который заполняется в случае, если в процессе обработки запроса произошла ошибка. Клиент, анализируя сообщение, содержащее soap:Fault, может узнать код ошибки, а также ее описание.

Далее представлено описание содержимого блока soap:Fault:

Элемент	Описание
<faultCode>	Код ошибки, определенный стандартом SOAP. Возможные значения этого элемента представлены далее, в следующей таблице.
<faultString>	Краткое текстовое описание ошибки
<faultActor>	Описание того, где произошла ошибка. Может заполняться, например, в случае, если одно сообщение последовательно проходит через несколько сервисов, и необходимо знать, на каком этапе произошла исключительная ситуация.
<detail>	Детальное описание ошибки

SOAP Fault Codes:

Код ошибки	Описание
S:VersionMismatch	Используется в случае, если при анализе сообщения обнаружено неверное пространство имен элемента S:Envelope. Как правило, такое встречается, если сервис и клиент работают по разным версия протокола SOAP.

S:Client	Используется в случае, если клиентское сообщение сформировано некорректно.
S:Server	Используется в случае, если в процессе обработки сообщения произошла ошибка на серверной стороне.

Далее приведен пример сообщения, содержащего soap:Fault.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
     <SOAP-ENV:Fault>
     <faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
     <faultstring xsi:type="xsd:string">
          Failed to locate method (ValidateCreditCard) in class
          (examplesCreditCard) at /usr/local/ActivePerl-5.6/lib/
            site_perl/5.6.0/SOAP/Lite.pm line 1555.
        </faultstring>

      </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В данном случае, ошибка произошла из-за того, что клиент вызвал метод ValidateCreditCard, который не поддерживается веб-сервисом.

2.8.1. Пример реализации обработки ошибок

В данном разделе будет показано, как произвести улучшение ранее разработанного веб-сервиса, с целью обработки им исключительной ситуации. Рассмотрим ситуацию, когда клиент вызывает метод "getPersonsByName", не задавая при этом единственный параметр – имя человека. Пусть, данная ситуация является искоючительной, и, следовательно, ее нужно обработать. Результат данной обработки ошибки – формирование веб-сервиом SOAP-сообщения, содержащего soap:Fault, с указанием того,что имя не должно быть пустым.

При разработке веб-сервисов используется точно такой же подход к обработке ошибок, как и при разработке любого другого Java-приложения: определяется свой класс-наследник java.lang.Exception, и затем с помощью ключевого слова throw, осуществляется “выброс” исключения при наступлении исключительной ситуации.

Однако, данный подход в JAX-WS является строго стандартизированным и накладывает некоторые ограничения на класс-наследник java.lang.Exception.

1. Для того,чтобы в коде веб-сервиса выбросить исключение, его класс должен быть аннотирован @WebFault.
2. В данной аннотации в параметре faultBean должен быть указан другой JavaBean класс, экземпляр которого содержит подробное описание ошибки, а также связанные с нею объекты. В итоге, аннотация над классом-исключением может выглядеть следующим образом: @WebFault(faultBean = "packagename.FaultBean").
3. Класс-исключение должен определять 2 конструктора со следующими параметрами: String message, FaultBean fault, а также String message, PersonServiceFault fault, Throwable cause
4. Класс-исключение должен определять метод getFaultInfo(), которые возвращает связанный с ним экземплят FaultBean.
5. Класс FaultBean должен определять метод String getMessage().

Следует отметить, что строка, переданная в конструктор классу-исключению, будет использована как значение блока <faultString>, а строка, возвращаемая методом getMessage() у класса FaultBean будет использована при построении блока <detail>.

Еще раз акцентируем внимание на том, что вышеописанные правила не являются так называемыми “best practises”, а закреплены на уровне JAX-WS, и работоспособность механизма обработки ошибок гарантируется только в случае их исполнения.

Теперь, после того как описаны основные правила обработки ошибок JAX-WS, будет показана реализация обработки исключительной ситуации, описанной в начале раздела.

Основные классы реализации:

Класс	Ответственность
PersonWebService	Основной класс веб-сервиса. Обратить внимание на метод getPersonsByName.
IllegalNameException	Класс-наследник java.lang.Exception. Выбрасывается сервисом,

	в случае, если не задано имя человека для поиска.
PersonServiceFault	Класс для детального описания ошибки. Служит faultBean для IllegalNameException.

PersonWebService.java

```
package com.wishmaster.ifmo.ws.jaxws;

import com.wishmaster.ifmo.ws.jaxws.errors.PersonServiceFault;
import com.wishmaster.ifmo.ws.jaxws.errors.IllegalNameException;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName = "PersonService")
public class PersonWebService {

    @WebMethod(operationName = "getAllPersons")
    public List<Person> getAllPersons() {
        PostgreSQLDAO dao = new PostgreSQLDAO();
        return dao.getAllPersons();
    }

    @WebMethod(operationName = "getPersonsByName")
    public List<Person> getPersonsByName(@WebParam(name = "personName") String name) throws
IllegalNameException {
        if (name == null || name.trim().isEmpty()) {
            PersonServiceFault fault = PersonServiceFault.defaultInstance();
            throw new IllegalNameException("personName is not specified", fault);
        }
        PostgreSQLDAO dao = new PostgreSQLDAO();
        return dao.getPersonsByName(name);
    }

}
```

Код данного класса изменился незначительно. Теперь, в случае, если имя не задано, выбрасывается IllegalNameException.

IllegalNameException.java

```
package com.wishmaster.ifmo.ws.jaxws.errors;

import javax.xml.ws.WebFault;

@WebFault(faultBean = "com.wishmaster.ifmo.ws.jaxws.errors.PersonServiceFault")
public class IllegalNameException extends Exception {
```

```
    private static final long serialVersionUID = -6647544772732631047L;
    private final PersonServiceFault fault;

    public IllegalNameException(String message, PersonServiceFault fault) {
        super(message);
        this.fault = fault;
    }

    public IllegalNameException(String message, PersonServiceFault fault, Throwable cause) {
        super(message, cause);
        this.fault = fault;
    }

    public PersonServiceFault getFaultInfo() {
        return fault;
    }
}
```

Данный класс – наследник java.lang.Exception, аннотированный @WebFault. В качестве faultBean указан класс PersonServiceFault. Обратите внимание на наличие двух конструкторов, а также метода getFaultInfo().Их необходимость обусловлена стандартом JAX-WS.

PersonServiceFault.java

```
package com.wishmaster.ifmo.ws.jaxws.errors;

public class PersonServiceFault {

    private static final String DEFAULT_MESSAGE = "personName cannot be null or empty";

    protected String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public static PersonServiceFault defaultInstance() {
        PersonServiceFault fault = new PersonServiceFault();
        fault.message = DEFAULT_MESSAGE;
        return fault;
    }

}
```

Класс, используемый в качестве faultBean для IllegalNameException. Обратите внимание на наличие метода getMessage().

Следует отметить, что если скомпилировать приложение и осуществить запрос к сервису через SoapUI, то ответ будет содержать полный stacktrace исключения. Зачастую такое поведение является нежелательным, и для того, чтобы избавиться от этого следует внести изменения в main-class приложения.

App.java

```
package com.wishmaster.ifmo.ws.jaxws;

import javax.xml.ws.Endpoint;

public class App {

    public static void main(String[] args) {
        //disable stacktraces in soap-message
        System.setProperty("com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace",
"false");
        String url = "http://0.0.0.0:8080/PersonService";
        Endpoint.publish(url, new PersonWebService());
    }
}
```

Свойство com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace устанавливается в false для того, чтобы в блок <detail> не выводился stacktrace выбрасываемого исключения.

Остальные классы, по сравнению с предыдущей реализацией не изменились. Теперь же в случае отправки запроса, не содержащего параметр personName, от сервиса возвращается следующий ответ:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>personName is not specified</faultstring>
      <detail>
        <ns2:IllegalNameException xmlns:ns2="http://jaxws.ws.ifmo.wishmaster.com/">
          <message>personName cannot be null or empty</message>
        </ns2:IllegalNameException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Необходимо отметить, что теперь следует изменить код клиента сервиса, и обращение к нему обернуть в блок try-catch, а также добавить обработку добавленного исключения.

Глава 3. REST-архитектура

3.1. Введение в REST

В прошлой главе были рассмотрены SOAP-сервисы, а также основные способы их разработки с использованием JAX-WS. Сущность SOAP-сервисов заключается в RPC (Remote Procedure Call) – в удаленном вызове процедур. Разработчик клиента SOAP-сервиса, используя WSDL-описание, генерирует артефакты сервиса, однм из которых является его интерфейс. Затем, написанный клиент взаимодействует с сервисом в рамках этого интерфейса. В данной главе будет представлен иной подход к организации взаимодействия между компонентами, а именно – REST (RESTful).

REST (Representational State Transfer, передача репрезентативного состояния) – архитектурный стиль построения распределенной системы, в основе которого лежит понятие ресурса и его состояния. Ресурс имеет свой универсальный идентификатор (URI, unified resource identifier), используя который над данным ресурсом можно совершать различные действия, например CRUD (create, read, update, delete).

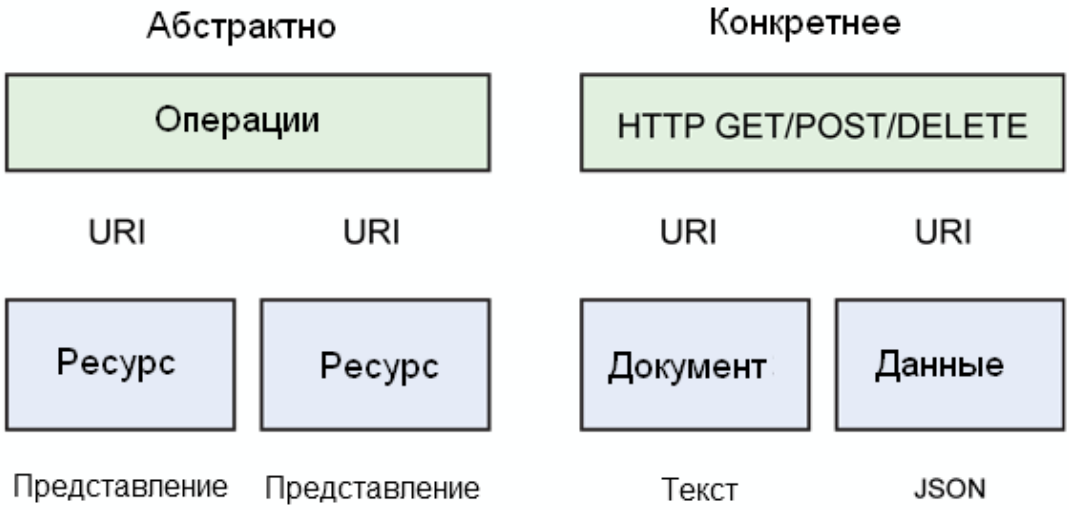


Рис. 1. REST-ресурс

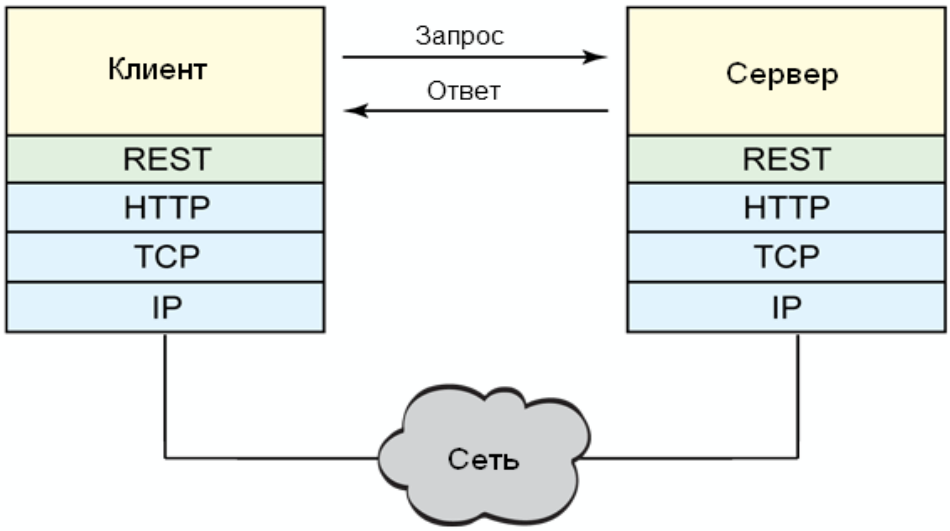


Рис. 2. Взаимодействие между клиентом и сервисом

Если речь идет о разработке REST-веб-сервисов, то, по сути, таким ресурсом и является веб-сервис, а взаимодействие производится не по протоколу SOAP, рассмотренному ранее, а по протоколу HTTP. Вследствие этого, набор возможных операций над ресурсом ограничен методами протокола HTTP.

Общее представление REST-ресурса представлено на рис.1, а схема взаимодействия клиента и сервиса на рис. 2. В отличие от сервисов, которые работают по протоколу SOAP, REST-сервисы не ограничены форматом XML для передачи данных, а могу использовать, например JSON, или plain-text.

Таким образом, REST — это не столько конкретная конструкция или реализация, сколько архитектурный стиль. Как показано на рисунке 1, архитектура RESTful определяется простым набором ограничений. В основе RESTful-архитектуры лежит набор ресурсов. Эти ресурсы определяются своим URI. Наконец, существует набор операций, с помощью которых этими ресурсами можно управлять. Этот набор ограничен методами протокола HTTP. Далее представлена таблица основных, используемых при реализации REST-архитектуры методов HTTP, с их кратким целевым назначением.

Метод	Описание
OPTIONS	Используется для запроса о поддерживаемых методах, адресах, а также дополнительной справочной информации
GET	Используется для извлечения состояния ресурса
HEAD	Используется для извлечения метаданных состояния ресурса
PUT	Создает или заменяет содержание ресурса
POST	Добавляет содержимое в существующий ресурс
DELETE	Используется для удаления ресурса

Следует отметить, что не обязательно использовать HTTP-методы только так, как указано в таблице. Вышеописанные случаи использования являются лишь рекомендациями, а также общепринятыми соглашениями.

Далее, приведем примеры REST-ресурсов, а также попытаемся спроектировать функциональность REST-сервиса, связанного с этими ресурсами. Пусть, предметной областью этих ресурсов будет библиотека. Предположим, что в данной библиотеке есть книги, а также есть пользователи, которые берут эти книги. Тогда у нас будут два корневых ресурса: первый – ресурс, представляющий пользователей, второй – книги соответственно. Положим, что мы хотим получить сервис, предоставляющий CRUD-функциональность для пользователей библиотеки и книг. Помимо этого требуется иметь подобные возможности и по отношению к книгам, взятым пользователем.

Далее в таблице представлены ресурсы, их URI, а также описания результатов различных вызовов по HTTP с различными методами:

URI	Описание	HTTP-метод	Результат вызова сервиса
/accounts	Корневой ресурс для представления	GET	Получение всех пользователей

	пользователей	POST	Создание нового пользователя
/accounts/{id}	Ресурс для представления конкретного пользователя	PUT	Изменение данных пользователя
		DELETE	Удаление пользователя
/books	Корневой ресурс для представления книг	GET	Получение всех книг
		POST	Создание новой книги
/books/{id}	Ресурс для преставления конкретной книги	PUT	Изменение данных книги
		DELETE	Удаление книги
/accounts/{id}/books	Ресурс для представления взятых книг конкретным пользователем	GET	Получение всех взятых книг конкретным пользователем
		POST	Добавление книги в коллекцию книг, взятых пользователем
/accounts/{a_id}/books{b_id}	Ресурс для изменения коллекции книг, взятых пользователем с id=a_id	DELETE	Удаление книги из коллекции книг, взятых пользователем

Из этой таблицы видно, что имея веб-сервис с такими ресурсами, мы сможем успешно решить поставленные задачи: можно добавлять, редактировать и удалять книги и пользователей, а также вести учет взятых конкретным пользователем книг. Но почему выбраны именно такие ресурсы и HTTP-методы?

Перед тем как перейти к рассмотрению вышеописанных ресурсов, следует заметить, что отдельно взятый ресурс может представлять, как отдельный объект, так и коллекцию объектов. Например “/accounts” представляет собой коллекцию пользователей, в то время как “/accounts/{id}” – один объект – пользователь с конкретным id. При проектировании этого REST API учитывалось, что операции update и delete объекта должны логически относиться к самому объекту, а не к коллекции, в которой он находится.

Учитывая это, имеем: ресурсы “/accounts” и “/books” представляет коллекции пользователей и книг соответственно. Можно запрашивать их содержимое, а также добавлять новые элементы. Для изменения данных пользователя или книги следует использовать функциональность ресурсов “/accounts/{id}” и “/book/{id}”. Также,

используя их, можно произвести удаление элемента по его id. Аналогичная концепция используется и в отношении ресурса “/accounts/{id}/books”.

Обратите внимание на то, что для создания нового объекта используется метод POST, в то время как метод PUT служит для изменения уже существующего объекта.

Хочется обратить внимание на то, что данные задачи можно было решить и по-другому, а именно, используя другой набор ресурсов. Однако выбран именно данный вариант в силу его простоты, а также непротиворечивости.

В отличие от SOAP-сервисов, REST-сервисы не имеют WSDL-описания и XSD-схем, что может значительно усложнить разработку клиента. При создании клиента, как правило, приходится вручную собирать запрос. Например, при создании новой книги, если бы это был SOAP-сервис, то разработчик создал бы новый экземпляр, например, класса Book, и с помощью set-методов заполнил поля объекта, а затем вызвал метод интерфейса. В случае REST-сервиса разработчику пришлось бы сначала указывать URI ресурса, а затем устанавливать некоторые значения HTTP-запроса. Таким образом, с какой-то точки зрения, разработка клиента является более долгой и трудоемкой.

3.1. Введение в JAX-RS

JAX-RS (Java API for RESTful Web Services) - это API в Java, обеспечивающий простое и быстрое создание REST-сервисов, который предлагает модель описания распределенных ресурсов на основе аннотаций, при помощи которых описываются местоположение и представление ресурсов. JAX-RS является частью Java EE, начиная с 6ой версии.

Подобно JAX-WS, JAX-RS предоставляет разработчику способы простой, гибкой разработки REST-сервисов. Несмотря на то, что JAX-RS входит в состав платформы Java EE, использовать это API можно и при разработке на Java SE.

Далее, как и в главе посвященной JAX-WS, будет показан процесс разработки двух вариантов REST-сервиса и клиента этого сервиса. Однако в отличие от SOAP-сервисов, REST-сервисы не всегда требуют специальных программных средств для их тестирования. В силу того, что REST-сервисы работают по протоколу HTTP, для тестирования GET-запросов можно использовать веб-браузер. Утилита SoapUI, которая использовалась для тестирования SOAP-сервисов, также имеет достаточно мощный функционал и может быть очень удобна для тестирования REST-сервисов.

Предметная область разрабатываемого сервиса будет полностью совпадать с предметной областью SOAP-сервиса из прошлой главы, и поэтому код для подключения к базе данных, таблицы в базе данных можно оставить без изменений.

3.2. Реализация сервиса с помощью JAX-RS

В данном разделе будет показано, как с помощью JAX-RS реализовать REST-сервис, который по функционалу будет аналогичен ранее разработанному SOAP-сервису. Сервис должен возвращать содержимое таблицы Persons базы данных

в формате JSON. Далее представлена таблица его основных классов, их реализация, а также основные моменты реализации, на которые следует обратить внимание.

Основные классы реализации:

Класс	Ответственность
App	Содержит main-метод, осуществляет запуск сервиса
Person	POJO, соответствует сущности, описанной в таблице persons базы данных
PersonResource	Класс, реализующий REST-ресурс

App.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;
import com.sun.jersey.api.core.ClassNamesResourceConfig;
import com.sun.jersey.api.core.ResourceConfig;
import java.io.IOException;
import java.net.URI;
import org.glassfish.grizzly.http.server.HttpServer;

public class App {

    private static final URI BASE_URI = URI.create("http://localhost:8080/rest/");
    public static void main(String[] args) {
        HttpServer server = null;
        try {
            ResourceConfig resourceConfig = new ClassNamesResourceConfig(PersonResource.class);
            server = GrizzlyServerFactory.createHttpServer(BASE_URI, resourceConfig);
            server.start();
            System.in.read();
            stopServer(server);
        } catch (IOException e) {
            e.printStackTrace();
            stopServer(server);
        }
    }

    private static void stopServer(HttpServer server) {
        if (server != null)
            server.stop();
    }
}
```

Предназначение данного класса полностью совпадает с предназначением класс App, используемого при реализации SOAP-сервиса. Его задача – запустить HTTP сервер и развернуть на нем REST-ресурс. После запуска сервис будет доступен по URL, заданному в константе BASE_URI.

По данному коду видно, что экземпляр класса HttpServer создается с помощью фабрики GrizzlyServerFactory. Метод для его создания принимает два параметра. Первый из них – базовый URI, по которому после запуска будут доступны REST-ресурсы, а второй – экземпляр класса ResourceConfig, который содержит информацию о тех ресурсах, которые необходимо развернуть на созданном сервере. В данном случае ресурс указан по его классу.

Вызов метода start объекта server осуществляет запуск сервера, и, в случае, если запуск произведен успешно, сервис сможет обрабатывать запросы. Следует отметить, что строка System.in.read(); необходима для того, чтобы сервер сразу же не завершал свою работу. А для того, чтобы остановить сервер необходимо ввести любой символ.

Person.java

```
package com.wishmaster.ifmo.ws.jaxrs.client;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Person {

    private String name;
    private String surname;
    private int age;

    public Person() {
    }

    public Person(String name, String surname, int age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", surname=" + surname + ", age=" + age + '}';
    }

}
```

Класс Person не претерпел существенных изменений. Единственное его отличие от предыдущих версий в том, что теперь он аннотирован @XmlRootElement. Данная JAXB-аннотация необходима для его корректного маршаллинга REST-сервисом.

Несмотря на то, что в нашем случае REST-сервис возвращает данные в формате JSON, используется аннотация @XmlRootElement (XML). Этот факт может сбить с толку. Это в первую очередь связано с тем, что поддержка REST-сервисов платформой Java появилась позже, чем JAXB, и, как следствие, JAXB-аннотации. Поэтому разработчики JAX-RS решили сохранить совместимость в JAXB, и для этого сделали поддержку JAXB-аннотаций, независимо от формата. Таким образом, один и те же аннотации могут использоваться, как для маршаллинга в формат XML, так и JSON.

PersonResource.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import java.util.List;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Path("/persons")
@Produces({MediaType.APPLICATION_JSON})
public class PersonResource {

    @GET
    public List<Person> getPersons(@QueryParam("name") String name) {
        List<Person> persons = new PostgreSQLDAO().getPersonsByName(name);
        return persons;
    }
}
```

Данный класс представляет собой REST-ресурс. Как было сказано ранее, конфигурирование производится с помощью аннотаций. @Path обозначает URI данного ресурса, относительно базового URI, который указывался при создании сервера в классе App. @Produces указывает формат данных, в котором сервис возвращает результат выполнения. Следует отметить, что этими двумя аннотациями может быть аннотирован как класс, так и метод. При аннотировании ими метода их смысл остается прежним.

@GET обозначает, что метод getPersons должен быть вызван при получении HTTP GET-запроса от клиента. При этом, в соответствии с @QueryParam("name"), из запроса извлекается значение параметра name и передается в метод в качестве аргумента. Следует отметить, что помимо @GET, в JAX-RS есть одноименные аннотации на каждый из методов протокола HTTP. И, аннотированный метод @POST, будет вызван для каждого POST-запроса по URL данного ресурса.

Обратите внимание на то, что, как и в случае с SOAP-сервисами, метод getPersons возвращает коллекцию объектов. При этом маршаллинг этих объектов в необходимый формат, а также анмаршаллинг входных параметров метода остается прозрачным для разработчика.

Если после компиляции и запуска данного приложения с помощью веб-браузера обратиться по http://localhost:8080/rest/persons, то результат будет следующим:


```
{"person": [
    {
    "age": "25",
    "name": "Петр",
    "surname": "Петров"
  },
    {
    "age": "26",
    "name": "Владимир",
    "surname": "Иванов"
  },
    {
    "age": "27",
    "name": "Иван",
    "surname": "Иванов"
  },
    {
    "age": "28",
    "name": "Иммануил",
    "surname": "Кант"
  },
    {
    "age": "29",
    "name": "Джордж",
    "surname": "Клуни"
  },
    {
    "age": "30",
    "name": "Билл",
    "surname": "Рубцов"
```

```
  },
    {
    "age": "31",
    "name": "Марк",
    "surname": "Марков"
  },
    {
    "age": "32",
    "name": "Галина",
    "surname": "Матвеева"
  },
    {
    "age": "33",
    "name": "Святослав",
    "surname": "Павлов"
  },
    {
    "age": "34",
    "name": "Ольга",
    "surname": "Бергольц"
  },
    {
    "age": "35",
    "name": "Лев",
    "surname": "Рабинович"
  }
]}
```


Как видно, сервис вернул по данному запросу все записи из базы данных. Однако в методе ресурса предусмотрена передача параметра name, который будет интерпретироваться, как имя человека при поиске. Для того, чтобы проверить работоспособность этого механизма, обратимся с помощью браузера по http://localhost:8080/rest/persons?name=Петр. В результате будет получен следующий результат:


```
{"person": {
  "age": "25",
  "name": "Петр",
  "surname": "Петров"
}}
```


Следовательно, REST-сервис работает корректно.

3.3. Реализация клиента

В отличие от SOAP-сервиса, REST-сервис не имеет WSDL-описания. Следовательно, генерация артефактов сервиса невозможна. А это значительно сказывается на процессе разработки клиента для него. В данном разделе будет показано, как реализовать клиент для ранее разработанного REST-сервиса.

Основные классы реализации:

Класс	Ответственность
App	Содержит main-метод и код для обращения к REST-сервису.

App.java

```
package com.wishmaster.ifmo.ws.jaxrs.client;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.GenericType;
import com.sun.jersey.api.client.WebResource;
import java.util.List;
import javax.ws.rs.core.MediaType;

public class App {

    private static final String URL = "http://localhost:8080/rest/persons";

    public static void main(String[] args) {
        Client client = Client.create();
        printList(getAllPersons(client, null));
        System.out.println();
        printList(getAllPersons(client, "Владимир"));
    }

    private static List<Person> getAllPersons(Client client, String name) {
        WebResource webResource = client.resource(URL);
        if (name != null) {
            webResource = webResource.queryParam("name", name);
        }
        ClientResponse response =
webResource.accept(MediaType.APPLICATION_JSON).get(ClientResponse.class);
        if (response.getStatus() != ClientResponse.Status.OK.getStatusCode()) {
            throw new IllegalStateException("Request failed");
        }
        GenericType<List<Person>> type = new GenericType<List<Person>>() {};
        return response.getEntity(type);
    }

    private static void printList(List<Person> persons) {
        for (Person person : persons) {
            System.out.println(person);
        }
    }
}
```

Одним из основных классов клиента для REST-сервиса является Client, создаваемый методом Client.create(). После получения экземпляра класса Client, с помощью него следует получить экзепмляр класса WebResource, передав в метод resource строковое представление URI. Если это необходимо, то параметры запроса

можно выставить с помощью webResource.queryParam(). Первый аргумент этого метода – название параметра, второй – его значение. С помощью класса WebResource также осуществляется обращение к сервису. Для этого методом accept() указывается ожидаемый формат, в котором данные будут получены от сервиса, а затем вызывается метод, соответствующий методу протокола HTTP. То есть для осуществления GET-метода следует вызвать метод get(), для POST – post(). В результате этой цепочки вызовов будет получен экземпляр класса ClientResponse, из которого можно извлечь код статус запроса, соответствующий статусу протокола HTTP (список кодов состояния протокола HTTP можно найти в стандарте протокола HTTP: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html), а также методом getEntity() – данные, которые вернул сервис. Обратите внимание на использование GenericType<List<Person>. GenericType необходимо использовать для анмаршаллинга generic-сущностей.

Сравнивая код клиента REST-сервиса с кодом клиента SOAP-сервиса, можно заключить, что отсутствие описания у REST-сервиса приводит к тому, что разработчик клиента должен обладать знаниями про следующие аспекты:

- URI всех используемых ресурсов
- Названия и возможные значения всех параметров запросов
- Формат данных используемый сервисом
- Назначение полей, возвращаемых сервисом

Из этого следует, что разработчик сервиса должен сделать удобную и в достаточной мере полную документацию по своему сервису. В противном случае разработка клиента может стать невозможной.

3.4. Реализация J2EE REST-сервиса

Подобно JAX-WS, JAX-RS обеспечивает универсальность и независимость от окружения. Различия между standalone сервисом и J2EE сервисом лишь в том, что последний не нуждается в коде, который будет запускать HTTP-сервер, поскольку это будет выполнено сервером приложений. Следовательно, отличия J2EE REST-сервиса от standalone REST-сервиса точно такие же, как и в случае с SOAP-сервисами.

Далее показана реализация REST-сервиса под платформу J2EE.

Класс	Ответственность
PersonResource	Класс, реализующий REST-ресурс

PersonResource.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import java.sql.Connection;
import java.sql.SQLException;
```

```
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.sql.DataSource;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@RequestScoped
@Path("/persons")
@Produces({MediaType.APPLICATION_JSON})
public class PersonResource {

   @Resource(lookup = "jdbc/ifmo-ws")
   private DataSource dataSource;

   @GET
   public List<Person> getPersons(@QueryParam("name") String name) {
      List<Person> persons = new PostgreSQLDAO(getConnection()).getPersonsByName(name);
      return persons;
   }

   private Connection getConnection() {
      Connection result = null;
      try {
         result = dataSource.getConnection();
      } catch (SQLException ex) {
         Logger.getLogger(PersonResource.class.getName()).log(Level.SEVERE, null, ex);
      }
      return result;
   }
}
```

Отличия этой версии ресурса от standalone-реализации лишь в том, что в данной версии происходит инъекция источника данных аннотацией @Resource. Затем этот источник данных используется для получения подключения к базе данных. В этом заключается универсальность JAX-RS.

Следует обратить внимание, что в силу реализации JAX-RS, при реализации J2EE REST-сервиса следует внести изменения в стандартный дескриптор развертывания.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
   <servlet>
      <servlet-name>ifmo-jax-rs-servlet</servlet-name>
      <servlet-class>
```

```
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
            <param-value>com.wishmaster.ifmo.ws.jaxrs</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ifmo-jax-rs-servlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Благодаря этому будет осуществляться вызов методов, определенных в ресурсе, при осуществлении запросов по /rest/*.

После развертывания этого сервиса, при попытке GET-запроса возвращается следующий результат:

```
{"person": [
    {
    "age": "25",
    "name": "Петр",
    "surname": "Петров"
  },
    {
    "age": "26",
    "name": "Владимир",
    "surname": "Иванов"
  },
    {
    "age": "27",
    "name": "Иван",
    "surname": "Иванов"
  },
    {
    "age": "28",
    "name": "Иммануил",
    "surname": "Кант"
  },
    {
    "age": "29",
    "name": "Джордж",
    "surname": "Клуни"
  },
    {
    "age": "30",
    "name": "Билл",
    "surname": "Рубцов"
  },
    {
    "age": "31",
```

```
    "name": "Марк",
    "surname": "Марков"
  },
    {
    "age": "32",
    "name": "Галина",
    "surname": "Матвеева"
  },
    {
    "age": "33",
    "name": "Святослав",
    "surname": "Павлов"
  },
    {
    "age": "34",
    "name": "Ольга",
    "surname": "Бергольц"
  },
    {
    "age": "35",
    "name": "Лев",
    "surname": "Рабинович"
  }
]}
```


Полученный результат свидетельствует о корректной работе REST-сервиса.

3.5. Обработка ошибок

JAX-RS, как и JAX-WS, предлагает разработчику достаточно гибкий механизм обработки ошибок. Но из-за отсутствия протокола SOAP, этот механизм не отягощен различными дополнительными правилами, и не накладывает никаких ограничений на состав и название методов.

Основная суть метода обработки ошибок в JAX-RS заключается в следующем: разработчик определяет свой обработчик собственного исключения и регистрирует в окружении JAX-RS. После успешной регистрации обработчика, при наступлении исключительной ситуации в коде ресурса достаточно лишь выбросить экземпляр зарегистрированного исключения с помощью ключевого слова throw. Все остальное JAX-RS сделает автоматически.

Рассмотрим такой же пример реализации обработки ошибок, который рассматривали и в SOAP-сервисах. В случае, если при поиске человека по имени имя не задано – выбросим исключение IllegalNameException.

Класс	Ответственность
IllegalNameException	Собственный класс исключения
IllegalNameExceptionMapper	Реализация интерфейса ExceptionMapper, который и

	представляет собой универсальный обработчик ошибок, описанный выше
App	Содержит main-метод, осуществляет запуск сервиса, регистрирует обработчик ошибок
PersonResource	Класс, реализующий REST-ресурс

IllegalNameException.java

```
package com.wishmaster.ifmo.ws.jaxrs;

public class IllegalNameException extends Exception {

   private static final long serialVersionUID = -6647544772732631047L;
   public static IllegalNameException DEFAULT_INSTANCE = new
IllegalNameException("personName cannot be null or empty");

   public IllegalNameException(String message) {
      super(message);
   }
}
```

Данный класс просто является наследником класса java.lang.Exception, и содержит один конструктор, в который принимает строку, содержащую текст сообщения об ошибке.

IllegalNameExceptionMapper.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class IllegalNameExceptionMapper implements ExceptionMapper<IllegalNameException> {

   @Override
   public Response toResponse(IllegalNameException e) {
      return Response.status(Status.BAD_REQUEST).entity(e.getMessage()).build();
   }
}
```

Данный класс реализует интерфейс ExceptionMapper, который и является обработчиком ошибок в JAX-RS. Класс типизирован тем исключением, которое мы собираемся обрабатывать. Интерфейс ExceptionMapper определяет единственный метод, возвращающий экземпляр класса Response, представляющий собой ответ от

ресурса. При реализации данного метода, в зависимости от самостоятельно определенного класса исключения, разработчик должен вручную собрать экземпляр класса Response и вернуть его. В данном случае, выставляется статус ответа, сигнализирующий, о том, что пользовательский запрос был некорректен, а также устанавливается тело ответа, в которое помещается строка с текстом ошибки.

Обратите внимание на аннотацию @Provider над классом. Она нужна для регистрации данного обработчика ошибок в JAX-RS среде.

App.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;
import com.sun.jersey.api.core.PackagesResourceConfig;
import com.sun.jersey.api.core.ResourceConfig;
import java.io.IOException;
import java.net.URI;
import org.glassfish.grizzly.http.server.HttpServer;

public class App {

   private static final URI BASE_URI = URI.create("http://localhost:8080/rest/");
   public static void main(String[] args) {
      HttpServer server = null;
      try {
         ResourceConfig resourceConfig = new
PackagesResourceConfig(PersonResource.class.getPackage().getName());
         server = GrizzlyServerFactory.createHttpServer(BASE_URI, resourceConfig);
         server.start();
         System.in.read();
         stopServer(server);
      } catch (IOException e) {
         e.printStackTrace();
         stopServer(server);
      }
   }

   private static void stopServer(HttpServer server) {
      if (server != null)
         server.stop();
   }
}
```

Реализация этого класса претерпела только одно изменение. Теперь экземпляр класса ResourceConfig, который используется для настройки ресурсов при создании сервере заменен на PackagesResourceConfig. Если ранее перечислялись классы, которые нужно зарегистрировать, то теперь указывается имя пакета, в котором эти классы находятся. И JAX-RS из этого пакета выбирает только специальным образом аннотированные классы для регистрации.

PersonResource.java

```
package com.wishmaster.ifmo.ws.jaxrs;

import java.util.List;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Path("/persons")
@Produces({MediaType.APPLICATION_JSON})
public class PersonResource {

   @GET
   public List<Person> getPersons() {
      return new PostgreSQLDAO().getPersonsByName(null);
   }

   @Path("/{name}")
   @GET
   public List<Person> getPersonsByName(@PathParam("name") String name) throws
IllegalNameException {
      if (name == null || name.trim().isEmpty())
         throw IllegalNameException.DEFAULT_INSTANCE;

      return new PostgreSQLDAO().getPersonsByName(name);
   }

}
```

В данной реализации в ресурс добавлен новый метод getPersonsByName, который по GET-запросу на URL /persons/{name} возвращает всех людей, у которых имя совпадает со значением {name}. В случае, если это имя не установлено или пустое, бросается IllegalNameException.

При попытке осуществить на http://localhost:8080/rest/persons/%20 GET-запрос, сервис возвращает следующий результат:

personName cannot be null or empty

Из этого можно заключить, обработка ошибок реализована так, как и предполагалось, и работает верно. Вообще, хочется отметить, что в JAX-RS обработка ошибок реализуется проще, нежели в JAX-WS. С какой-то точки зрения в JAX-RS она интуитивно понятна и не требует запоминания стандарта.

Глава 4. SOAP или REST сервисы?

В предыдущих главах были рассмотрены базовые принципы SOAP и REST-сервисов, а также некоторые аспекты их реализации. Основная задача данной главы заключается в том, чтобы показать разницу между SOAP и REST сервисами.

Перед тем, как перейти к рассмотрению достоинств и недостатков той или иной стороны, хотелось бы отметить, что SOAP, по сути, является универсальным RPC-протоколом, в то время как REST – архитектурный подход к разработке программных систем, в центре которого ставится понятие ресурса. Поэтому сравнивать эти два понятия друг с другом не совсем корректно. В данной же главе будет показано, чем будут отличаться сервисы REST от SOAP.

Основные моменты, отличающие REST и SOAP, представлены далее:

1. SOAP активно использует XML для кодирования запросов и ответов, а также строгую типизацию данных, гарантирующую их целостность при передаче между клиентом и сервером. С другой стороны, запросы и ответы в REST могут передаваться в ASCII, XML, JSON или любых других форматах, распознаваемых одновременно и клиентом, и сервером. Кроме того, в модели REST отсутствуют встроенные требования к типизации данных. В результате пакеты запросов и ответов в REST имеют намного меньшие размеры, чем соответствующие им пакеты SOAP.
2. В модели SOAP уровень передачи данных протокола HTTP является «пассивным наблюдателем», и его роль ограничивается передачей запросов SOAP от клиента серверу с использованием метода POST. Детали сервисного запроса, такие как имя удаленной процедуры и входные аргументы, кодируются в теле запроса. Архитектура REST, напротив, рассматривает уровень передачи данных HTTP как активного участника взаимодействия, используя существующие методы HTTP, такие как GET, POST, PUT и DELETE, для обозначения типа запрашиваемого сервиса. Следовательно, с точки зрения разработчика, запросы REST в общем случае более просты для формулирования и понимания, так как они используют существующие и хорошо понятные интерфейсы HTTP.
3. Модель SOAP поддерживает определенную степень интроспекции, позволяя разработчикам сервиса описывать его API в файле формата WSDL. Используя их, клиенты SOAP могут автоматически получать подробную информацию об именах и сигнатурах методов, типах входных и выходных данных и возвращаемых значениях. Модель REST не имеет WSDL, однако во многом ее интерфейс может быть более понятен и даже интуитивен, так как основан на стандартных методах HTTP.
4. В основе REST лежит концепция ресурсов, в то время как SOAP использует интерфейсы, основанные на объектах и методах. Интерфейс SOAP может содержать практически неограниченное количество методов; интерфейс REST, напротив, ограничен четырьмя возможными операциями, соответствующими четырем методам HTTP.

Упомянутая в п.1 разница в размерах пакетов иногда может быть весьма значительной. Для примера далее представлены запрос и ответ SOAP и REST, от сервиса rpc.geocoder.us.

SOAP-запрос:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
                   xmlns:ns1="http://rpc.geocoder.us/Geo/Coder/US/"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:xsl="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
                   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:geocode>
      <location xsl:type="xsd:string">1600 Pennsylvania Av, Washington, DC</location>
    </ns1:geocode>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP-ответ:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:xsl="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <geocodeREspose xmlns="http://rpc.geocoder.us/Geo/Coder/US/">
      <geo:results soapenc:arrayType="geo:GeocoderAddressResult[1]"
                   xsl:type="soapenc:Array"
                   xmlns:geo="http://rpc.geocoder.us/Geo/Coder/US/">
        <geo:item                                    xsl:type="geo:GeocoderAddressResult"
xmlns:geo="http://rpc.geocoder.us/Geo/Coder/US/">
          <geo:number xsl:type="xsd:int">1600</geo:number>
          <geo:lat xsl:type="xsd:float">38.898748</geo:lat>
          <geo:street xsl:type="xsd:string">Pensylvania</geo:street>
          <geo:state xsl:type="xsd:string">DC</geo:state>
          <geo:city xsl:type="xsd:string">Washington</geo:city>
          <geo:zip xsl:type="xsd:int">20502</geo:zip>
          <geo:suffix xsl:type="xsd:string">NW</geo:suffix>
          <geo:long xsl:type="xsd:float">-77.037684</geo:long>
          <geo:type xsl:type="xsd:string">Ave</geo:type>
          <geo:prefix xsl:type="xsd:string">
        </geo:item>
      </geo:results>
    </geocodeResponse>
  </soap:Body>
</soap:Envelope>
```

REST-запрос:

```
GET http://geocoder.us/service/rest/geocode?address=1600+Pennsylvania+Ave,+Washington+DC
```

REST-ответ:

```
<?xml version="1.0"?>
<rdf:RDF
   xmls:dc="http://purl.org/dc/elements/1.1/"
   xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <geo:Point rdf:nodeID="aid47091944">
      <dc:description>1600 Pensylvaia Ave NW, Washington DC 20502</dc:description>
      <geo:long>-77.037684</geo:long>
      <geo:lat>38.898748</geo:lat>
   </geo:point>
</rdf:RDF>
```

В обоих случаях вызывалась различная реализация одного и того же сервиса. То насколько меньше и компактнее запрос и ответ REST-сервиса, видно невооруженным взглядом. Из-за этого на поддержание SOAP-сервисов требуются бОльшие вычислительные мощности, а объем сетевого трафика значительно больше при их использовании.

Помимо п.1 особенно хочется отметить п.3. Действительно, невозможность генерации артефактов по WSDL усложняет жизнь, как разработчику сервиса, так и разработчику клиента. Разработчик сервиса, зная, что не сможет доставить описание сервиса до клиента, вынужден сделать документацию, в достаточной мере полную, чтобы можно было написать клиентское приложение. Разработчик клиента должен потратить достаточное количество времени на освоение этой документации, а также на тестирование своего клиента, поскольку ручная сборка запросов и парсинг ответов часто бывают подвержены различным ошибкам.

Однако все же хочется понять, когда какие сервисы лучше использовать? Но, на этот вопрос не существует однозначного ответа. Скорее всего, необходимо отталкиваться от реально поставленных задач, а универсального решения просто не существует.

Например, если речь идет о разработке системы корпоративного уровня, особенно, с возможными потребностями в интеграции с другими системами, то, скорее всего, следует сделать выбор в сторону SOAP-сервисов. Во-первых, это будет оправдано, потому что SOAP-сервисы имеют мощный потенциал к интеграции с помощью ESB. Во-вторых, как правило, число платформ, с использованием которых построена корпоративная система невелико, и, следовательно, нет требования поддержать возможность взаимодействия с множеством других платформ.

Обратная ситуация наблюдается в случае, если необходимо разработать сервис, ориентированный на большое число пользователей, и в данном случае выгоднее использовать REST. Во-первых, это связано с широкой поддержкой протокола HTTP современными платформами и языками программирования. Конечно, протокол SOAP тоже поддерживается, но часто встречаются случаи, когда простая разработка клиента на платформе, отличной от той, на которой разработан сервис, просто невозможна. Например, не удается по WSDL сгенерировать артефакты, или происходят ошибки во время маршаллинга-анмаршаллинга объектов. Во-вторых, с точки зрения производительности выгоднее REST. Возможно, именно поэтому крупные социальные сети, такие как ВКонтакте, Facebook, Twitter предлагают именно REST API, а не SOAP API.

Следует отметить, что к выбору между SOAP или REST следует подходить с большой ответственностью, и не следует его делать без тщательного анализа, хотя бы, потому что миграция с SOAP на REST и наоборот не является легкой в силу колоссальной разницы между REST и RPC.

Глава 6. Реестры сервисов

6.1. UDDI-реестры

Как было показано в главе 1, реестр сервисов является крайне важным звеном сервис-ориентированной архитектуры. Ведь именно в нем хранится информация обо всех зарегистрированных сервисах, и он используется для поиска сервисов для взаимодействия.

Базовый стандарт для построения реестров сервисов - UDDI (Universal Description, Discovery and Integration) – стандарт, позволяющий публиковать и находить сервисы. Фактически, UDDI-реестр является специализированным приложением, используя функциональность которого, поставщики сервисов регистрируют свои сервисы, а потенциальные клиенты ищут наиболее подходящие для них. Как правило, такие системы имеют возможности классификации и категоризации, зарегистрированных в них сервисов.

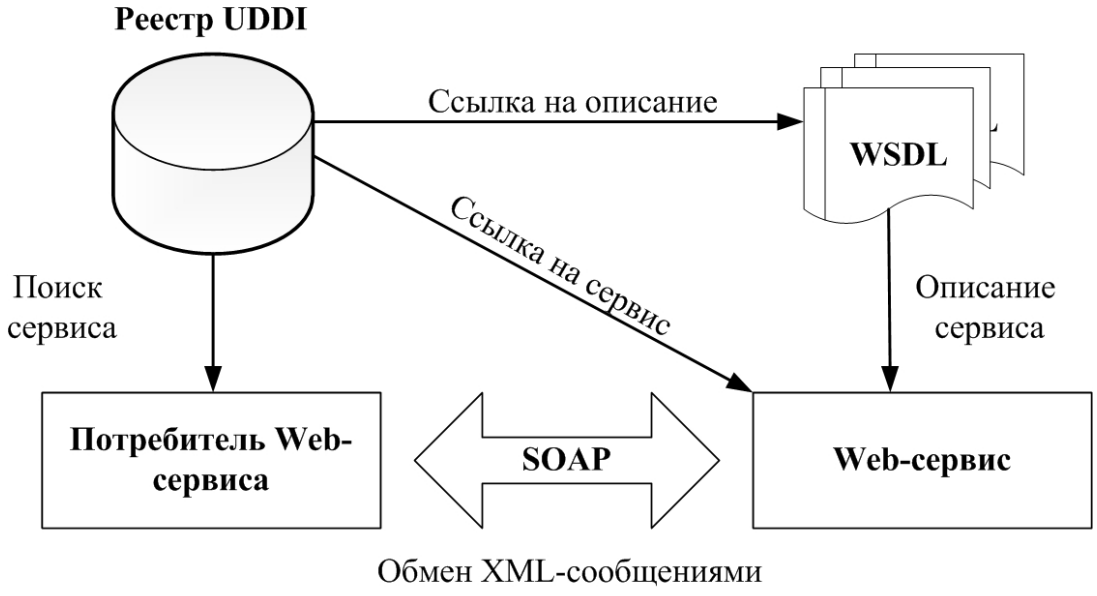


Рис. 1.2. Организация реестра UDDI

Важно отметить, что информацию из реестра можно условно разделить на три группы по характеру данной информации: белые страницы, желтые страницы и зеленые страницы. Белые страницы характеризую поставщика сервиса, предоставляя его описание, контактные данные и другую личную информацию. Желтые страницы описывают сервисы по сферам применения, а также каталогизируют и классифицируют их. Зеленые страницы несут в себе техническую информацию, необходимую для обращения к сервисам.

Белые страницы	Желтые страницы	Зеленые страницы
Описывают поставщика сервиса: • Информация об организации • Контактные данные • Географическое расположение	Описывают сервис, исходя из его сферы применения: • Предметная область • Сфера деятельности	Техническая информация о сервисе: • Доступность • Время ответа • Требования к клиенту сервиса • Безопасность • WSDL • XSD

Таким образом, поиск можно проводить по различным критериям, начиная от местоположения поставщика услуг и, заканчивая особенностями интерфейса сервиса.

Реестр UDDI является наиболее распространенной разновидностью поискового реестра. Благодаря тому, что он логически разделен на три типа каталогов: белые, желтые и зеленые страницы, в нем можно искать подходящий сервис как по контактным данным поставщика, так по сфере применения, а также и по техническим деталям, обуславливающим обращение к сервису. Поэтому предприятие А из предыдущего примера может найти в нем будущего партнера по бизнесу.

Однако поиск по реестру UDDI является несколько ограниченным, так как предоставляет лишь условия поиска отдельных сервисов. Скорее всего, для большинства предприятий, которые ищут нового партнера, данные условия поиска будут удовлетворительны. Но, например, тем организациям, которые намереваются

или же расширить спектр предоставляемых услуг, или же освоить новый род деятельности будут необходимы более широкие критерии поиска. Например, было бы удобно искать партнеров по описаниям бизнес-процессов, в которых они участвуют. Такую концепцию поиска предлагает ebXML.

6.2. ebXML

ebXML (Electronic Business using eXtensible Markup Language) – стандарт, обеспечивающий инфраструктуру обмена бизнес-информацией и создание общей бизнес-среды для различных компаний. Данный стандарт позволяет с помощью XML создавать описания бизнес-процессов и хранить их в реестре, давая возможность осуществлять поиск информации о бизнес-партнерах.

Таким образом, ebXML предлагает более широкие возможности, чем хранение и поиск информации о веб-сервисах в корпоративной среде. Ведь речь уже идет о создании единого электронного рынка. Особенно эффективным будет использование ebXML-реестра для консорциума, объединяющего множество компаний.

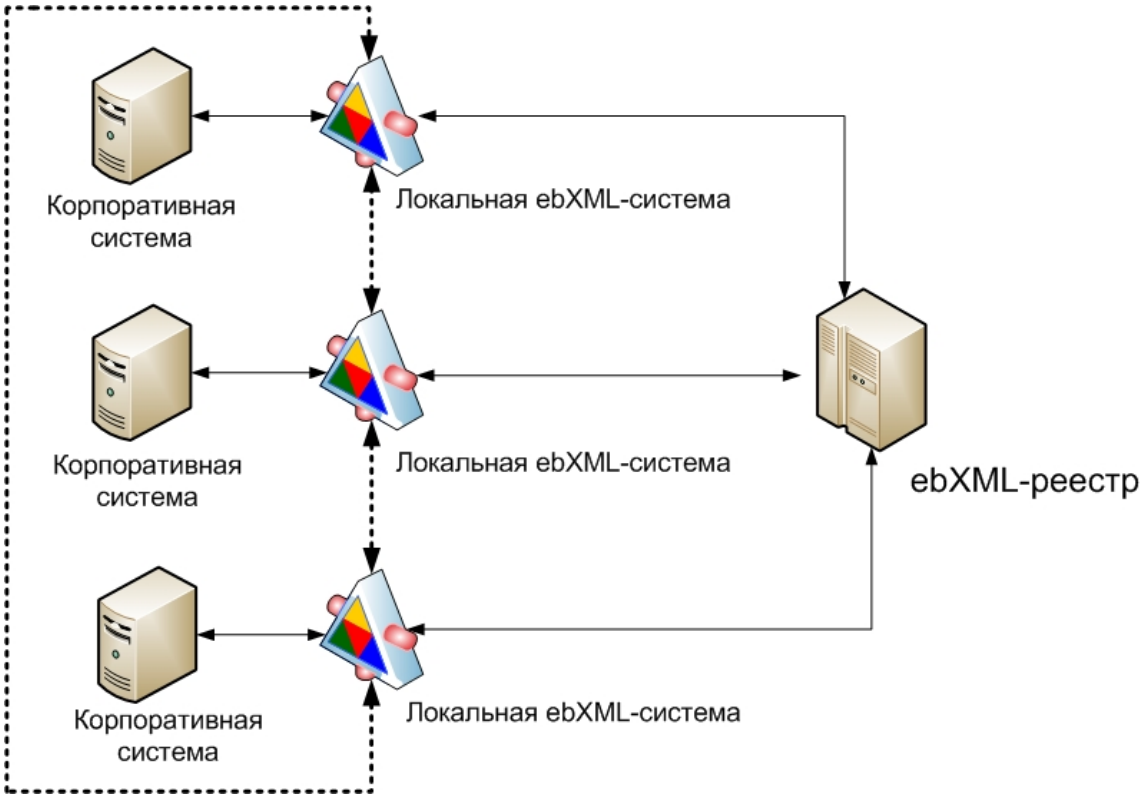


Рис. 2.4. Организация взаимодействия с использованием ebXML-реестра

Концепция ebXML предполагает наличие центрального реестра. Фактически, данный реестр является информационной системой, в которой реализована совокупность веб-сервисов для осуществления взаимодействия с клиентами. Компания-пользователь работает с этим реестром через локальную ebXML-систему, реализованную в ее собственной корпоративной системе. В состав локальной системы также входит и клиент для взаимодействия с сервисами центрального реестра. Однако помимо клиент-серверного взаимодействия с реестром локальные ebXML-системы компаний также и взаимодействуют друг с другом, что и является, на мой взгляд, наиболее существенным отличием от ранее рассмотренного UDDI.

Стандартный сценарий использования ebXML состоит из следующих этапов.

1) Компания A решает присоединиться к использованию ebXML-реестра, и для этого она запрашивает у него требования к локальной ebXML-системе. После установки этой системы, информационная система компании сможет взаимодействовать с центральных ebXML-реестром, а также с другими участниками.
2) Компания A с помощью локальной ebXML-системы создает один или несколько профилей, которые в терминологии ebXML называются Collaboration Protocol Profile, или CPP-профиль. Данные профили описывают бизнес-процессы компании A, а также некоторый дополнительный набор данных, который будет анализироваться другими участниками для принятия решения о дальнейшем сотрудничестве. Например, такими данными могут быть географическое расположение компании, контактные данные и прочая информация подобного характера. Созданные CPP-профили регистрируются в центральном ebXML-реестре.
3) Компания B с помощью своей локальной ebXML-системы находит в реестре сведения о компании A. И если после анализа профиля компании A она заинтересуется участием в ее бизнес-процессах, то на основе документов профилей двух компаний генерируется новый документ – Collaboration Protocol Agreement, который, по сути, является договором. Компания B посылает этот договор компании A, и если последняя его

утвердит, то данный документ будет передан обратно компании B, и далее будет использоваться для конфигурирования обеих локальных систем предприятий. После того, как эти систему будут сконфигурированы, две компании смогут обмениваться бизнес-документами и следовать бизнес-процессам в соответствии с CPA-документом.

Таким образом, ebXML предлагает достаточно удобную схему взаимодействия между многими предприятиями.

Если проводить сравнение ebXML с UDDI, то можно сказать, что UDDI в первую очередь, является service-oriented, а ebXML – business-process-oriented. ebXML эффективнее использовать для организации межкорпоративного взаимодействия, когда круг потенциальных партнеров заранее неизвестен, в то время как UDDI лучше подойдет для организации реестра сервисов в рамках небольшого числа предприятий хорошо известных друг другу. В силу достаточно сложной организации ebXML является менее распространенным, чем UDDI.

6.3. Типы реестров

Исходя из модели взаимодействия, реестры сервисов можно условно разделить на 2 основные группы: Design-time and Runtime. Описанная выше модель поиска, которая предоставляется UDDI-реестрами специфична для Design-time реестров.

Design-time реестры предоставляют в конечном итоге лишь функционал для поиска сервиса, и конечный артефакт, получаемый в процессе поиска – описания искомого сервиса. Далее, на основании этих описаний, разработчик генерирует клиент для веб-сервиса, а также код, осуществляющий к нему обращения. То есть поиск организуется благодаря функционалу реестра, а взаимодействия – благодаря разработчику.

Runtime-реестры позволяют не только найти сервис, но и также осуществить к нему обращение, без необходимости ручной генерации клиента, а также написания кода. Также Runtime-реестры обладают мощными возможностями обнаружения сервисов, интерфейсы которых удовлетворяют определенным описаниям.

Также Runtime-реестр имеет возможности для композиции сервисов, с целью получения необходимого результата. Например, клиенту необходимо, совершив одно обращение к удаленному сервису, получить набор данных A и B. При этом, набор данных A – возвращается только сервисом A`, а B – B`, и нет возможности получить оба набора за одно обращение. Для решения данной проблемы, Design-time реестр, после анализа поискового запроса, произведет поиск сервисов для получения наборов A и B, осуществит к ним обращение, а затем вернет клиенту требуемые наборы данным. Таким образом, с точки зрения клиента, было совершено лишь одно обращение к удаленным сервисам. Таким образом, Design-time реестры обладают средствами поддержки принятия решений, а также средствами композиции сервисов для получения требуемого результата.

Далее в таблице представлены характеристики реестров обоих типов.

Design-time	Runtime
• Human interactive • Позволяет получить WSDL искомого сервиса • Обычно, поиск осуществляется через веб-интерфейс	• Machine-to-machine • Focus on discovery of endpoints that comply with certain service interfaces • Имеет средства поддержки принятия решений • Композиция сервисов для получения нужного результата

Следует отметить, что, вследствие того, что функциональность Runtime реестра значительно превосходит лишь возможности хранения, а также поиска информации о сервисах, то Runtime реестр можно рассматривать, как совокупность Design-time-реестра, а также некого интеграционного ПО, например, сервисной шины.

В силу значительно более сложного внутреннего устройства, Runtime реестры менее распространены, и встречаются крайне редко.

6.4. Существующие реестры

Наиболее известные реестры сервисов представлены далее:

- Hewlett Packard – Systinet
- Apache jUDDI
- WSMX (в составе)
- WSO2 (в составе)
- Mule ESB (в составе)

Реестр Systinet – проприетарный реестр, разработанный компанией Hewlett Packard. Как правило, используется крупными компаниями и корпорациями, вследствие высокой производительности, надежности, а также широкой поддержки, предоставляемой разработчиком.

Рис. 6.1. Поиск сервиса Systinet

На рис. 6.1. представлен скриншот графического интерфейса Systinet, для ввода поискового запроса. При вводе поискового запроса возможно использование специальных символов, таких как ‘%’ и ‘*’.

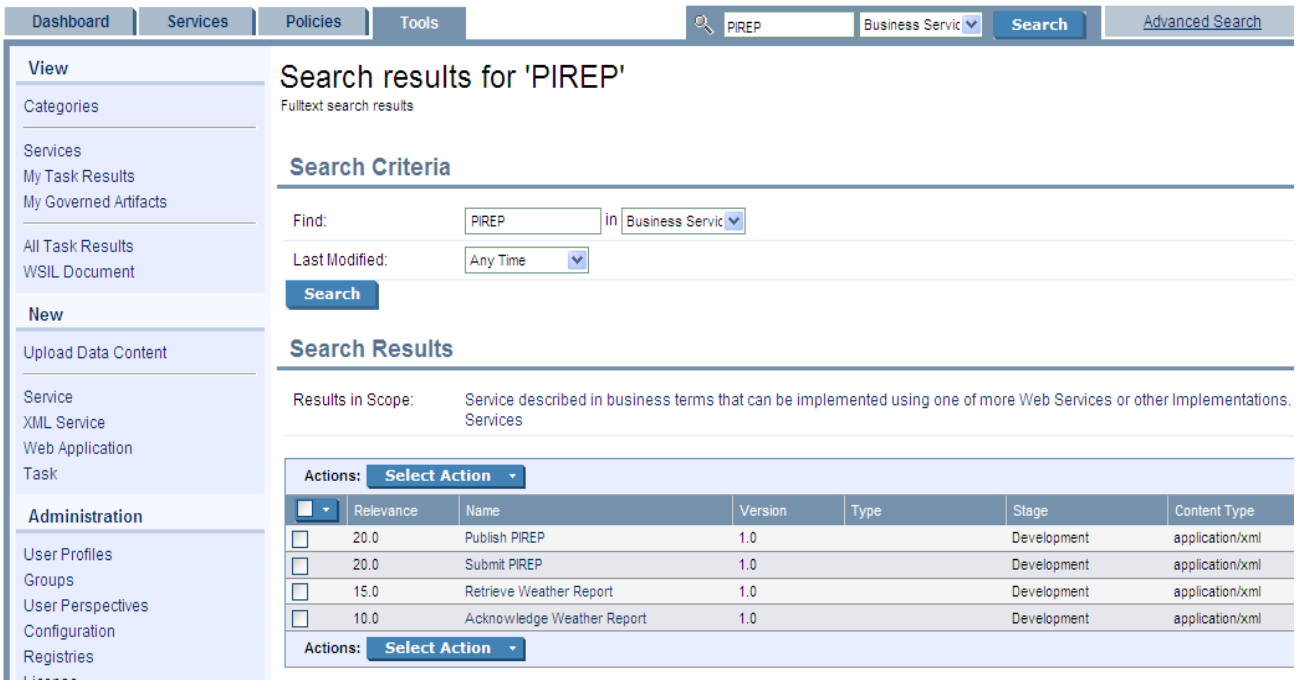


Рис. 6.2. Отображение результатов поиска

На рис. 6.2. представлен скриншот, на котором показано отображение результатов поиска сервисов. В таблице ‘Search Results’, отображено название

сервиса, версия, стадия жизненного цикла, на которой он в данный момент находится, а также Content-Type, возвращаемых этим сервисом данных.

На рис. 6.3. представлен скриншот, на котором отображается детальная информация по выбранному сервису. Среди дополнительной отображаемой информации следует отметить наличие таких полей, как ключевые слова, владелец сервиса, каналы доставки (обуславливают способы соединения с этим сервисом; Internet – веб-сервис доступен из сети Интернет, Private Network – только для хостов из определенной локальной сети).

Рис. 6.3. Детальная информация о выбранном сервисе

Apache jUDDI – один из немногих активно развивающихся реестров с открытым исходным кодом. Релизы Apache jUDDI проходят примерно 3-5 раз в год. Благодаря этому, у этого реестра есть множество потребителей, а также сообщество разработчиков, которые используют этот реестр.

Продукты WSMX, WSO2, Mule ESB являются комплексными интеграционными платформами, в составе которых также присутствуют и реестры сервисов.

Существуют также и публичные реестры сервисов. Наиболее известные среди них:

- programmableweb.com
- service-repository.com

- webservicex.net

6.5. Apache jUDDI

В данном разделе описан процесс установки, а также некоторые ключевые моменты при работе с реестром сервисов Apache jUDDI. При разработке данного пособия, а также заданий использовалась версия 3.2.0, релиз которой состоялся 05.02.2014.

6.5.1. Установка и запуск

Apache jUDDI – свободно-распространяемый реестр сервисов. Скачать его, документацию, видео уроки, а также найти сообщество разработчиков можно по адресу juddi.apache.org. Архив с версией 3.2.0 рекомендуется скачать по данному адресу http://apache-mirror.rbc.ru/pub/apache/juddi/juddi/3.2.0/.

Скачанный архив следует разархивировать. В каталоге “examples” содержится большое количество примеров исходного кода для работы с данным реестром. Данные примеры следует использовать при выполнении лабораторной работы. Для запуска следует воспользоваться скриптом “startup.sh” или “startup.bat”, расположенным в “juddi-tomcat-3.2.0/bin”.

Если запуск прошел без ошибок, то при переходе по ссылке http://localhost:8080/juddi-gui/ будет отображен графический браузерный клиент реестра, как показано на скриншоте далее.

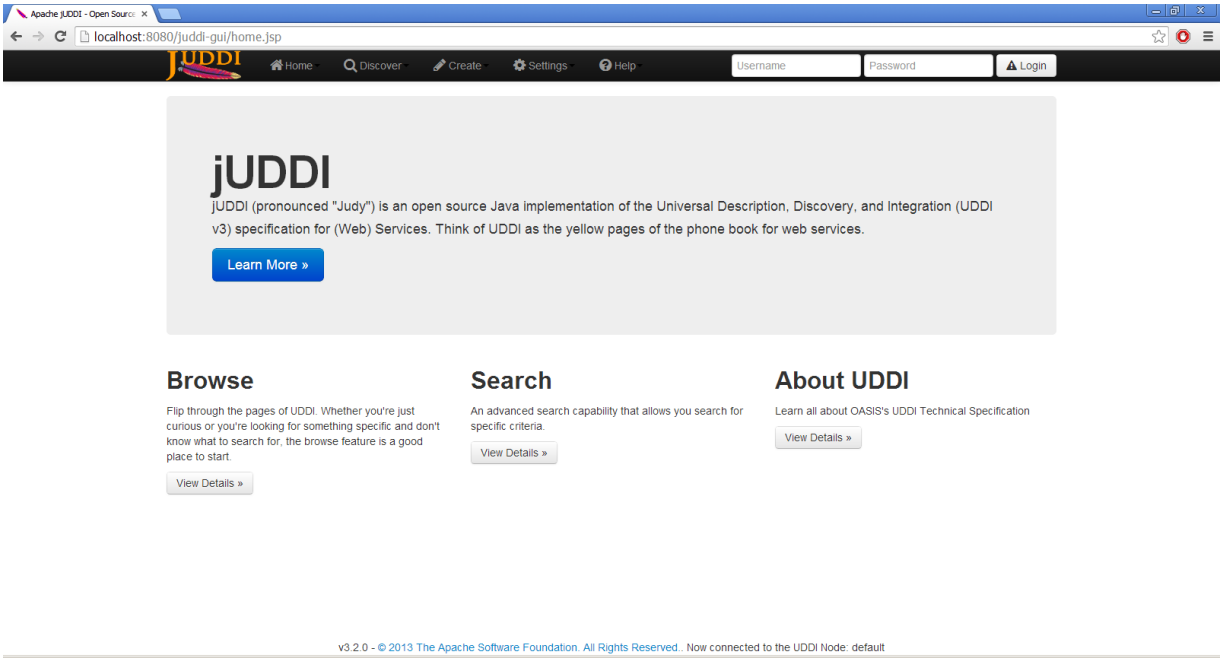


Рис. 6.5.1. Веб-клиент jUDDI

Чтобы авторизоваться, следует использовать открыть файл tomcat-users.xml, и ввести логин и пароль пользователя с ролью “uddiadmin”. Если в данном файле отсутствует запись пользователя с ролью uddiadmin, то ее следует добавить и перезапустить jUDDI. Например, запись в данном файле может выглядеть так:

```
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="uddiadmin"/>
  <user username="uddiadmin" password="da_password1" roles="uddiadmin,tomcat,manager" />
</tomcat-users>
```

После добавления этого XML в файле tomcat-users.xml и перезапуска jUDDI станет возможной авторизация с логином “uddiadmin” и паролем “da_password1”.

6.5.2. Регистрация бизнеса

Непосредственно перед регистрацией сервиса в jUDDI, необходимо указать информацию о поставщике и сфере применения сервиса, т.е. заполнить белые и желтые страницы. После заполнения этих данных, можно зарегистрировать сервис.

Первый этап, в терминологии jUDDI называется регистрацией бизнеса, а сущность, агрегирующая информацию по белым и желтым страницам – бизнесом. Для

регистрации бизнеса с помощью веб-клиента, следует выбрать пункт меню “create”, а затем подпункт “business”.

Далее на скриншоте представлен интерфейс регистрации бизнеса.

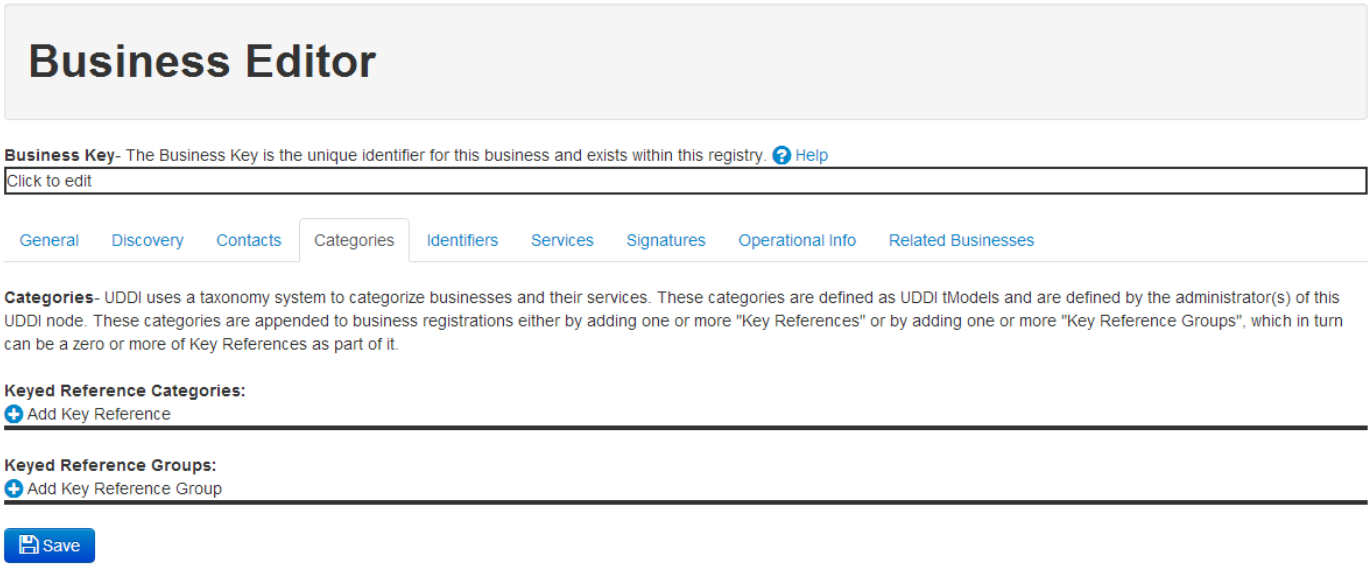


Рис. 6.5.2. Регистрация бизнеса

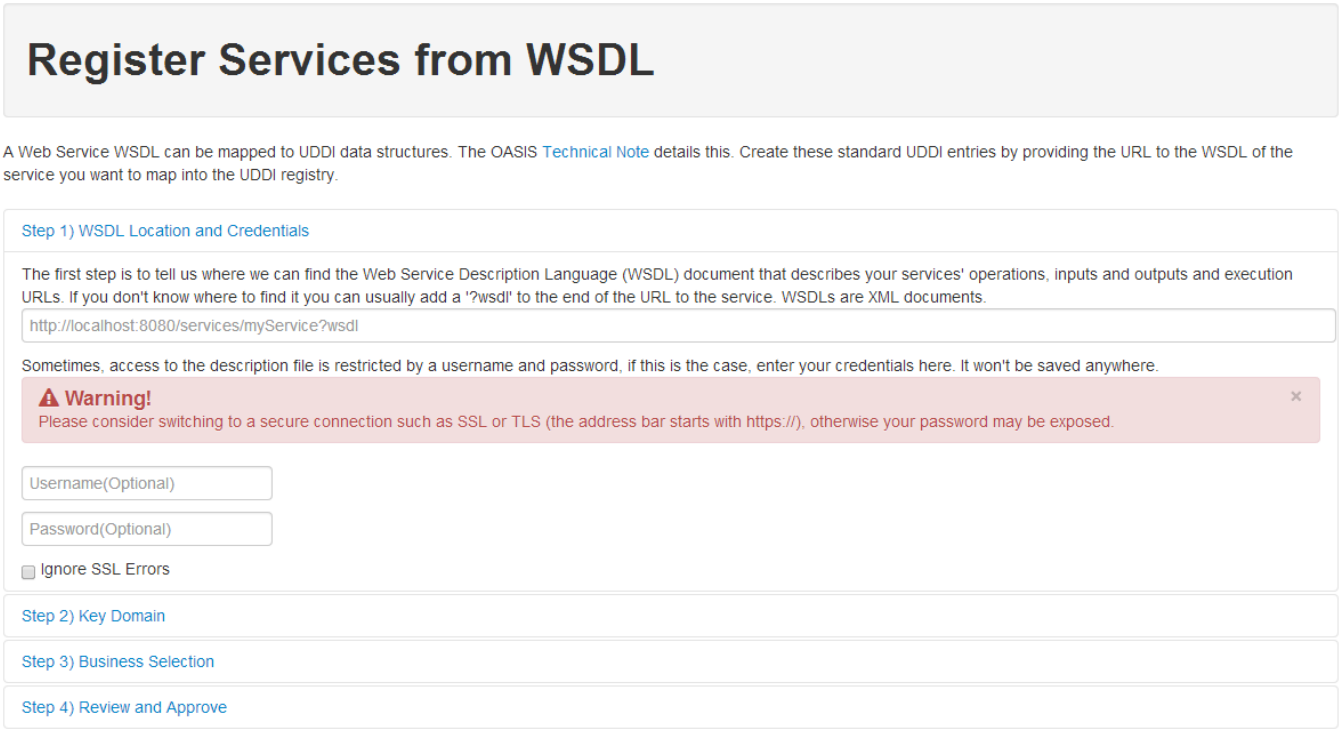
На различных вкладках содержится справочная информация, о том, как их следует заполнять. Основная информация, которую следует задать при регистрации бизнеса в jUDDI:

Раздел	Состав данных
General	Название и описание
Discovery	Ссылки на внешние источники с описаниями поставщика
Contacts	Контакты, такие как телефон, email, адрес
Categories	Ключевые слова, специфичные для данного бизнеса; например, сфера деятельности
Identifiers	Специальные идентификаторы, регистрационные номера поставщика
Signatures	Цифровые подписи поставщика
Related businesses	Связанные бизнесы с данным

6.5.3. Регистрация сервиса

После ввода необходимых данных, а также успешного сохранения бизнеса, найти данный бизнес и просмотреть информацию о нем можно, выбрав пункт меню “Discover”, а затем “Businesses”.

Затем, выбрав “Create” -> “Register services from WSDL”, следует провести регистрацию сервиса. Далее на скриншоте показан интерфейс регистрации сервиса.



Для регистрации сервиса следует указать расположение WSDL-описания сервиса, а затем указать бизнес, к которому относится данный сервис. В случае успешного доступа к WSDL по указанному адресу, сервис будет зарегистрирован, и так же его можно будет найти на вкладке “Discover”.

6.5.4. Поиск

На рис. 6.5.4. представлен скриншот интерфейса для поиск бизнеса или сервиса. Поиск можно осуществлять по полному или частичному соответствию, результаты поиска можно сортировать.

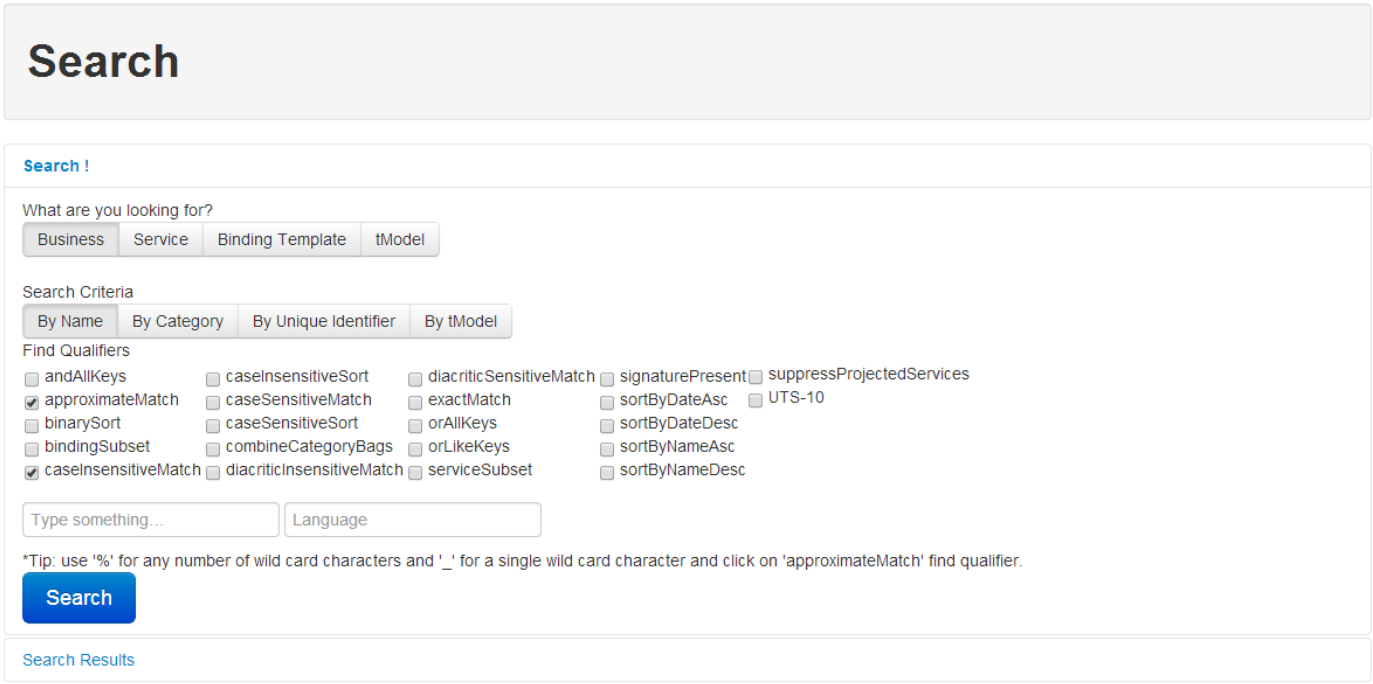


Рис. 6.5.4. Поиск

6.5.5. Программное взаимодействие с jUDDI

В данном разделе приведен пример поиска бизнесов и сервисов из реестра jUDDI.

Данный пример – проект SimpleBrowse из официальных примеров кода jUDDI, расположенных в каталоге “examples” архива.

SimpleBrowse.java

```
/*
 * Copyright 2001-2010 The Apache Software Foundation.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */
package org.apache.juddi.example.browse;

import java.util.List;
import org.apache.juddi.api_v3.AccessPointType;
import org.apache.juddi.v3.client.UDDIConstants;
import org.apache.juddi.v3.client.config.UDDIClient;
import org.apache.juddi.v3.client.transport.Transport;
import org.uddi.api_v3.AuthToken;
import org.uddi.api_v3.BindingTemplates;
import org.uddi.api_v3.BusinessDetail;
import org.uddi.api_v3.BusinessInfos;
import org.uddi.api_v3.BusinessList;
import org.uddi.api_v3.BusinessService;
import org.uddi.api_v3.CategoryBag;
import org.uddi.api_v3.Contacts;
import org.uddi.api_v3.Description;
import org.uddi.api_v3.DiscardAuthToken;
import org.uddi.api_v3.FindBusiness;
import org.uddi.api_v3.GetAuthToken;
import org.uddi.api_v3.GetBusinessDetail;
import org.uddi.api_v3.GetServiceDetail;
import org.uddi.api_v3.KeyedReference;
import org.uddi.api_v3.Name;
import org.uddi.api_v3.ServiceDetail;
import org.uddi.api_v3.ServiceInfos;
import org.uddi.v3_service.UDDIInquiryPortType;
import org.uddi.v3_service.UDDISecurityPortType;

/**
 * A Simple UDDI Browser that dumps basic information to console
 *
 * @author <a href="mailto:alexoree@apache.org">Alex O'Ree</a>
 */
public class SimpleBrowse {

        private static UDDISecurityPortType security = null;
        private static UDDIInquiryPortType inquiry = null;

        /**
         * This sets up the ws proxies using uddi.xml in META-INF
```

```
 */
public SimpleBrowse() {
	try {
	// create a manager and read the config in the archive;
		// you can use your config file name
		UDDIClient client = new UDDIClient("META-INF/simple-browse-uddi.xml");
	// a UDDIClient can be a client to multiple UDDI nodes, so
		// supply the nodeName (defined in your uddi.xml.
		// The transport can be WS, inVM, RMI etc which is defined in the uddi.xml
		Transport transport = client.getTransport("default");
		// Now you create a reference to the UDDI API
		security = transport.getUDDISecurityService();
		inquiry = transport.getUDDIInquiryService();
	} catch (Exception e) {
		e.printStackTrace();
	}
}

/**
 * Main entry point
 *
 * @param args
 */
public static void main(String args[]) {

	SimpleBrowse sp = new SimpleBrowse();
	sp.Browse(args);
}

public void Browse(String[] args) {
	try {

		String token = GetAuthKey("uddi", "uddi");
		BusinessList findBusiness = GetBusinessList(token);
		PrintBusinessInfo(findBusiness.getBusinessInfos());
		PrintBusinessDetails(findBusiness.getBusinessInfos(), token);
		PrintServiceDetailsByBusiness(findBusiness.getBusinessInfos(), token);

		security.discardAuthToken(new DiscardAuthToken(token));

	} catch (Exception e) {
		e.printStackTrace();
	}
}

/**
 * Find all of the registered businesses. This list may be filtered
 * based on access control rules
 *
 * @param token
 * @return
 * @throws Exception
 */
private BusinessList GetBusinessList(String token) throws Exception {
	FindBusiness fb = new FindBusiness();
	fb.setAuthInfo(token);
	org.uddi.api_v3.FindQualifiers fq = new org.uddi.api_v3.FindQualifiers();
	fq.getFindQualifier().add(UDDIConstants.APPROXIMATE_MATCH);
```

```
            fb.setFindQualifiers(fq);
            Name searchname = new Name();
            searchname.setValue(UDDIConstants.WILDCARD);
            fb.getName().add(searchname);
            BusinessList findBusiness = inquiry.findBusiness(fb);
            return findBusiness;

        }

        /**
         * Converts category bags of tmodels to a readable string
         *
         * @param categoryBag
         * @return
         */
        private String CatBagToString(CategoryBag categoryBag) {
            StringBuilder sb = new StringBuilder();
            if (categoryBag == null) {
                return "no data";
            }
            for (int i = 0; i < categoryBag.getKeyedReference().size(); i++) {
                sb.append(KeyedReferenceToString(categoryBag.getKeyedReference().get(i)));
            }
            for (int i = 0; i < categoryBag.getKeyedReferenceGroup().size(); i++) {
                sb.append("Key Ref Grp: TModelKey=");
                for (int k = 0; k < categoryBag.getKeyedReferenceGroup().get(i).getKeyedReference().size();
k++) {

sb.append(KeyedReferenceToString(categoryBag.getKeyedReferenceGroup().get(i).getKeyedReference().get(k)));
                }
            }
            return sb.toString();
        }

        private String KeyedReferenceToString(KeyedReference item) {
            StringBuilder sb = new StringBuilder();
            sb.append("Key Ref: Name=").
                append(item.getKeyName()).
                append(" Value=").
                append(item.getKeyValue()).
                append(" tModel=").
                append(item.getTModelKey()).
                append(System.getProperty("line.separator"));
            return sb.toString();
        }

        private void PrintContacts(Contacts contacts) {
            if (contacts == null) {
                return;
            }
            for (int i = 0; i < contacts.getContact().size(); i++) {
                System.out.println("Contact " + i + " type:" + contacts.getContact().get(i).getUseType());
                for (int k = 0; k < contacts.getContact().get(i).getPersonName().size(); k++) {
                    System.out.println("Name: " +
contacts.getContact().get(i).getPersonName().get(k).getValue());
                }
                for (int k = 0; k < contacts.getContact().get(i).getEmail().size(); k++) {
                    System.out.println("Email: " + contacts.getContact().get(i).getEmail().get(k).getValue());
                }
```

```
                    for (int k = 0; k < contacts.getContact().get(i).getAddress().size(); k++) {
                        System.out.println("Address sort code " +
contacts.getContact().get(i).getAddress().get(k).getSortCode());
                        System.out.println("Address use type " +
contacts.getContact().get(i).getAddress().get(k).getUseType());
                        System.out.println("Address tmodel key " +
contacts.getContact().get(i).getAddress().get(k).getTModelKey());
                        for (int x = 0; x < contacts.getContact().get(i).getAddress().get(k).getAddressLine().size();
x++) {
                            System.out.println("Address line value " +
contacts.getContact().get(i).getAddress().get(k).getAddressLine().get(x).getValue());
                            System.out.println("Address line key name " +
contacts.getContact().get(i).getAddress().get(k).getAddressLine().get(x).getKeyName());
                            System.out.println("Address line key value " +
contacts.getContact().get(i).getAddress().get(k).getAddressLine().get(x).getKeyValue());
                        }
                    }
                    for (int k = 0; k < contacts.getContact().get(i).getDescription().size(); k++) {
                        System.out.println("Desc: " +
contacts.getContact().get(i).getDescription().get(k).getValue());
                    }
                    for (int k = 0; k < contacts.getContact().get(i).getPhone().size(); k++) {
                        System.out.println("Phone: " + contacts.getContact().get(i).getPhone().get(k).getValue());
                    }
                }

            }

            private void PrintServiceDetail(BusinessService get) {
                if (get == null) {
                    return;
                }
                System.out.println("Name " + ListToString(get.getName()));
                System.out.println("Desc " + ListToDescString(get.getDescription()));
                System.out.println("Key " + (get.getServiceKey()));
                System.out.println("Cat bag " + CatBagToString(get.getCategoryBag()));
                if (!get.getSignature().isEmpty()) {
                    System.out.println("Item is digitally signed");
                } else {
                    System.out.println("Item is not digitally signed");
                }
                PrintBindingTemplates(get.getBindingTemplates());
            }

            /**
             * This function is useful for translating UDDI's somewhat complex data
             * format to something that is more useful.
             *
             * @param bindingTemplates
             */
            private void PrintBindingTemplates(BindingTemplates bindingTemplates) {
                if (bindingTemplates == null) {
                    return;
                }
                for (int i = 0; i < bindingTemplates.getBindingTemplate().size(); i++) {
                    System.out.println("Binding Key: " +
bindingTemplates.getBindingTemplate().get(i).getBindingKey());
                //TODO The UDDI spec is kind of strange at this point.
```

```
                    //An access point could be a URL, a reference to another UDDI binding key, a hosting redirector
(which is
                    //esscentially a pointer to another UDDI registry) or a WSDL Deployment
                    //From an end client's perspective, all you really want is the endpoint.
                    //http://uddi.org/pubs/uddi_v3.htm#_Ref8977716
                    //So if you have a wsdlDeployment useType, fetch the wsdl and parse for the invocation URL
                    //If its hosting director, you'll have to fetch that data from uddi recursively until the leaf node is
found
                    //Consult the UDDI specification for more information

                    if (bindingTemplates.getBindingTemplate().get(i).getAccessPoint() != null) {
                        System.out.println("Access Point: " +
bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getValue() + " type " +
bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType());
                        if (bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType() != null) {
                            if
(bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType().equalsIgnoreCase(AccessPointType.E
ND_POINT.toString())) {
                                System.out.println("Use this access point value as an invocation endpoint.");
                            }
                            if
(bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType().equalsIgnoreCase(AccessPointType.B
INDING_TEMPLATE.toString())) {
                                System.out.println("Use this access point value as a reference to another binding
template.");
                            }
                            if
(bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType().equalsIgnoreCase(AccessPointType.
WSDL_DEPLOYMENT.toString())) {
                                System.out.println("Use this access point value as a URL to a WSDL document,
which presumably will have a real access point defined.");
                            }
                            if
(bindingTemplates.getBindingTemplate().get(i).getAccessPoint().getUseType().equalsIgnoreCase(AccessPointType.
HOSTING_REDIRECTOR.toString())) {
                                System.out.println("Use this access point value as an Inquiry URL of another
UDDI registry, look up the same binding template there (usage varies).");
                            }
                        }
                    }
                }
            }

            private enum AuthStyle {

                HTTP_BASIC,
                HTTP_DIGEST,
                HTTP_NTLM,
                UDDI_AUTH,
                HTTP_CLIENT_CERT
            }

            /**
             * Gets a UDDI style auth token, otherwise, appends credentials to the
             * ws proxies (not yet implemented)
             *
             * @param username
             * @param password
             * @param style
```

```
     * @return
     */
    private String GetAuthKey(String username, String password) {
        try {

            GetAuthToken getAuthTokenRoot = new GetAuthToken();
            getAuthTokenRoot.setUserID(username);
            getAuthTokenRoot.setCred(password);

            // Making API call that retrieves the authentication token for the user.
            AuthToken rootAuthToken = security.getAuthToken(getAuthTokenRoot);
            System.out.println(username + " AUTHTOKEN = (don't log auth tokens!");
            return rootAuthToken.getAuthInfo();
        } catch (Exception ex) {
            System.out.println("Could not authenticate with the provided credentials " + ex.getMessage());
        }
        return null;
    }



    private void PrintBusinessInfo(BusinessInfos businessInfos) {
        if (businessInfos == null) {
            System.out.println("No data returned");
        } else {
            for (int i = 0; i < businessInfos.getBusinessInfo().size(); i++) {
                System.out.println("================================================");
                System.out.println("Business Key: " +
businessInfos.getBusinessInfo().get(i).getBusinessKey());
                System.out.println("Name: " +
ListToString(businessInfos.getBusinessInfo().get(i).getName()));

                System.out.println("Description: " +
ListToDescString(businessInfos.getBusinessInfo().get(i).getDescription()));
                System.out.println("Services:");
                PrintServiceInfo(businessInfos.getBusinessInfo().get(i).getServiceInfos());
            }
        }
    }

    private String ListToString(List<Name> name) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < name.size(); i++) {
            sb.append(name.get(i).getValue()).append(" ");
        }
        return sb.toString();
    }

    private String ListToDescString(List<Description> name) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < name.size(); i++) {
            sb.append(name.get(i).getValue()).append(" ");
        }
        return sb.toString();
    }

    private void PrintServiceInfo(ServiceInfos serviceInfos) {
        for (int i = 0; i < serviceInfos.getServiceInfo().size(); i++) {
            System.out.println("----------------------------------------");
```

```
                System.out.println("Service Key: " + serviceInfos.getServiceInfo().get(i).getServiceKey());
                System.out.println("Owning Business Key: " +
serviceInfos.getServiceInfo().get(i).getBusinessKey());
                System.out.println("Name: " + ListToString(serviceInfos.getServiceInfo().get(i).getName()));
            }
        }

        private void PrintBusinessDetails(BusinessInfos businessInfos, String token) throws Exception {
            GetBusinessDetail gbd = new GetBusinessDetail();
            gbd.setAuthInfo(token);
            for (int i = 0; i < businessInfos.getBusinessInfo().size(); i++) {
                gbd.getBusinessKey().add(businessInfos.getBusinessInfo().get(i).getBusinessKey());
            }
            BusinessDetail businessDetail = inquiry.getBusinessDetail(gbd);
            for (int i = 0; i < businessDetail.getBusinessEntity().size(); i++) {
                System.out.println("Business Detail - key: " +
businessDetail.getBusinessEntity().get(i).getBusinessKey());
                System.out.println("Name: " +
ListToString(businessDetail.getBusinessEntity().get(i).getName()));
                System.out.println("CategoryBag: " +
CatBagToString(businessDetail.getBusinessEntity().get(i).getCategoryBag()));
                PrintContacts(businessDetail.getBusinessEntity().get(i).getContacts());
            }
        }

        private void PrintServiceDetailsByBusiness(BusinessInfos businessInfos, String token) throws Exception {
            for (int i = 0; i < businessInfos.getBusinessInfo().size(); i++) {
                GetServiceDetail gsd = new GetServiceDetail();
                for (int k = 0; k <
businessInfos.getBusinessInfo().get(i).getServiceInfos().getServiceInfo().size(); k++) {

gsd.getServiceKey().add(businessInfos.getBusinessInfo().get(i).getServiceInfos().getServiceInfo().get(k).getServiceK
ey());
                }
                gsd.setAuthInfo(token);
                System.out.println("Fetching data for business " +
businessInfos.getBusinessInfo().get(i).getBusinessKey());
                ServiceDetail serviceDetail = inquiry.getServiceDetail(gsd);
                for (int k = 0; k < serviceDetail.getBusinessService().size(); k++) {
                    PrintServiceDetail(serviceDetail.getBusinessService().get(k));
                }
                System.out.println(".....");

            }
        }
    }
```

Глава 5. Задания для самостоятельного выполнения

5.1. Работа №1. Поиск с помощью SOAP-сервиса.

В данной работе требуется создать таблицу в БД, содержащую не менее 5 полей, а также реализовать возможность поиска по любым комбинациям полей с помощью SOAP-сервиса. Данные для поиска должны передаваться в метод сервиса в качестве аргументов.

Веб-сервис необходимо реализовать в виде standalone-приложения и J2EE-приложения. При реализации в виде J2EE-приложения следует на стороне сервера приложений настроить источник данных, и осуществлять его инъекцию в коде сервиса.

Для демонстрации работы разработанных сервисов следует также разработать и клиентское консольное приложение.

5.2. Работа №2. Реализация CRUD с помощью SOAP-сервиса.

В данной работе в веб-сервис, разработанный в первой работе, необходимо добавить методы для создания, изменения и удаления записей из таблицы.

Метод создания должен принимать значения полей новой записи, метод изменения – идентификатор изменяемой записи, а также новые значения полей, а метод удаления – только идентификатор удаляемой записи.

Метод создания должен возвращать идентификатор новой записи, а методы обновления или удаления – статус операции. В данной работе следует вносить изменения только в standalone-реализацию сервиса.

В соответствии с изменениями сервиса необходимо обновить и клиентское приложение.

5.3. Работа №3. Обработка ошибок в SOAP-сервисе.

Основываясь на информации из раздела 2.8, добавить поддержку обработки ошибок в сервис. Возможные ошибки, которые могут происходить при добавлении новых записей – например, неверное значение одного из полей, при изменении, удалении – попытка изменить или удалить несуществующую запись.

В соответствии с изменениями сервиса необходимо обновить и клиентское приложение.

5.4. Работа №4. Поиск с помощью REST-сервиса.

Необходимо выполнить задание из первой работы, но с использованием REST-сервиса. Таблицу базы данных, а также код для работы с ней можно оставить без изменений.

5.5. Работа №5. Реализация CRUD с помощью REST-сервиса.

Необходимо выполнить задание из второй работы, но с использованием REST-сервиса. Таблицу базы данных, а также код для работы с ней можно оставить без изменений.

5.6. Работа №6. Обработка ошибок в REST-сервисе.

Необходимо выполнить задание из третьей работы, но с использованием REST-сервиса. Таблицу базы данных, а также код для работы с ней можно оставить без изменений.

5.7. Работа №7. Регистрация и поиск сервиса в реестре jUDDI

Требуется разработать приложение, осуществляющее регистрацию сервиса в реестре jUDDI, а также поиск сервиса в реестре и обращение к нему. Рекомендуется реализовать консольное приложение, которое обрабатывает 2 команды. Итог работы первой команды – регистрация сервиса в реестре; вторая команда должна осуществлять поиск сервиса, а также обращение к нему.

Приложение

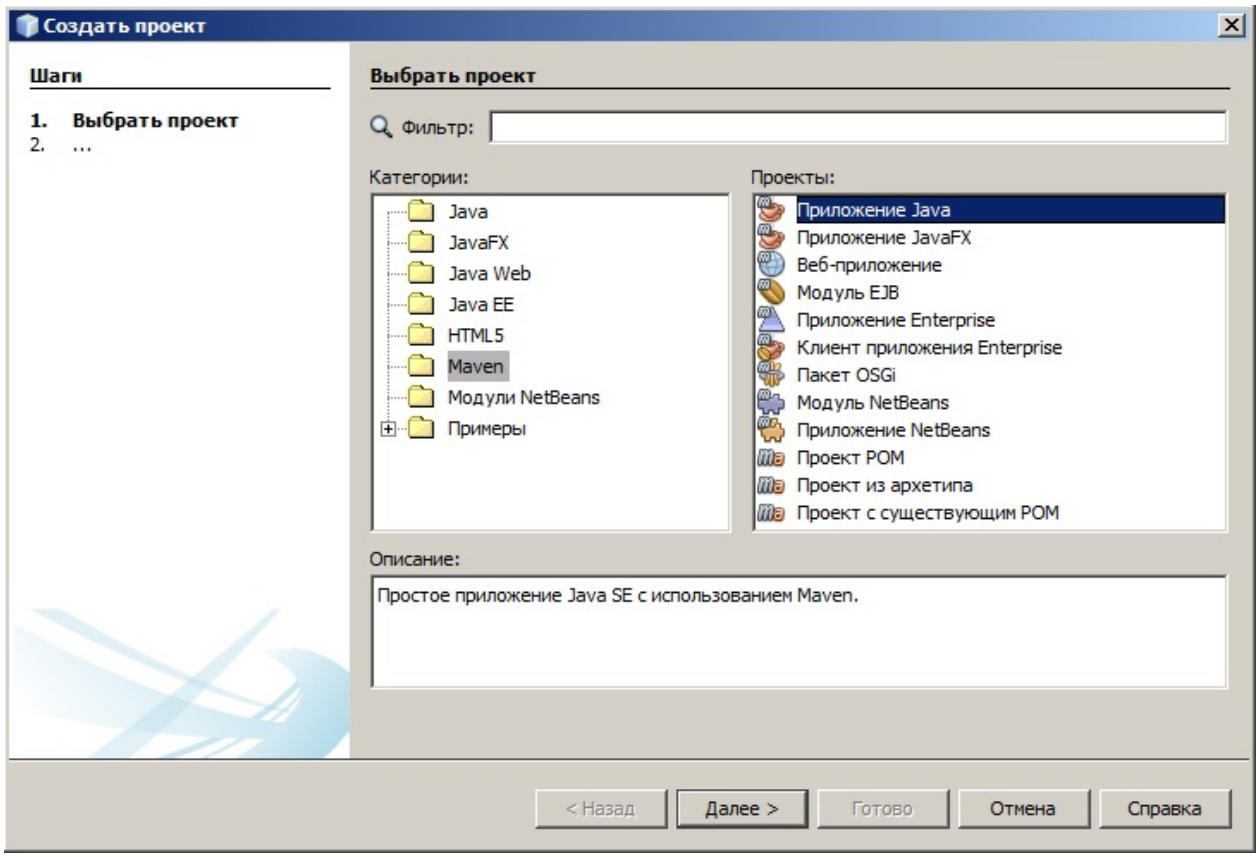
Maven

Для сборки сервисов в рамках данного курса использовалась система автоматизации сборки Apache Maven. Maven, в отличие от другого сборщика проектов Apache Ant, обеспечивает декларативную, а не императивную сборку проекта. То есть, в файлах проекта (pom.xml) содержится его декларативное описание, а не отдельные команды.

При использовании Maven, по сравнению с Ant, значительно облегчается сборка, поскольку в файле проекта pom.xml указываются необходимые для работы библиотеки (в терминологии Maven – зависимости), а уже Maven при сборке осуществляет загрузку этих зависимостей из доступных репозиториев.

На сайте http://maven.apache.org/ можно скачать Maven, а также найти документацию, а также полезные примеры. Также для изучения Maven хорошо подойдет эта статья: http://habrahabr.ru/post/77382/.

Следует отметить, что поддержка Maven встроена в современные среды разработки, такие как NetBeans, Intellij Idea, Eclipse. При использовании NetBeans только при создании проекта можно указать, что необходимо использовать Maven. Пример создания Maven проекта приведен на следующем скриншоте:



Необходимые для сборки зависимости Maven

Для реализации standalone SOAP-сервиса необходимо в pom.xml добавить следующие зависимости:

```
<dependency>
   <groupId>com.sun.xml.ws</groupId>
   <artifactId>webservices-rt</artifactId>
   <version>1.4</version>
   <scope>compile</scope>
</dependency>

<dependency>
   <groupId>com.sun.xml.bind</groupId>
   <artifactId>jaxb-impl</artifactId>
   <version>2.2.7</version>
</dependency>

<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>9.2-1003-jdbc4</version>
</dependency>
```

Необходимые зависимости для реализации J2EE-сервиса SOAP:

```
<dependency>
   <groupId>javax</groupId>
   <artifactId>javaee-web-api</artifactId>
   <version>7.0</version>
   <scope>provided</scope>
</dependency>
```

Для реализации JAX-RS standalone-сервиса необходимы следующие зависимости:

```
<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-server</artifactId>
   <version>1.17.1</version>
</dependency>

<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-servlet</artifactId>
   <version>1.17.1</version>
</dependency>

<dependency>
   <groupId>com.sun.jersey</groupId>
   <artifactId>jersey-grizzly2</artifactId>
   <version>1.17.1</version>
</dependency>

<dependency>
   <groupId>com.sun.jersey</groupId>
```

```
    <artifactId>jersey-json</artifactId>
    <version>1.17.1</version>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.2-1003-jdbc4</version>
</dependency>
```

Для JAX-RS J2EE сервиса:

```
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-server</artifactId>
    <version>1.17.1</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-grizzly2</artifactId>
    <version>1.17.1</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-servlet</artifactId>
    <version>1.17.1</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.17.1</version>
</dependency>
```

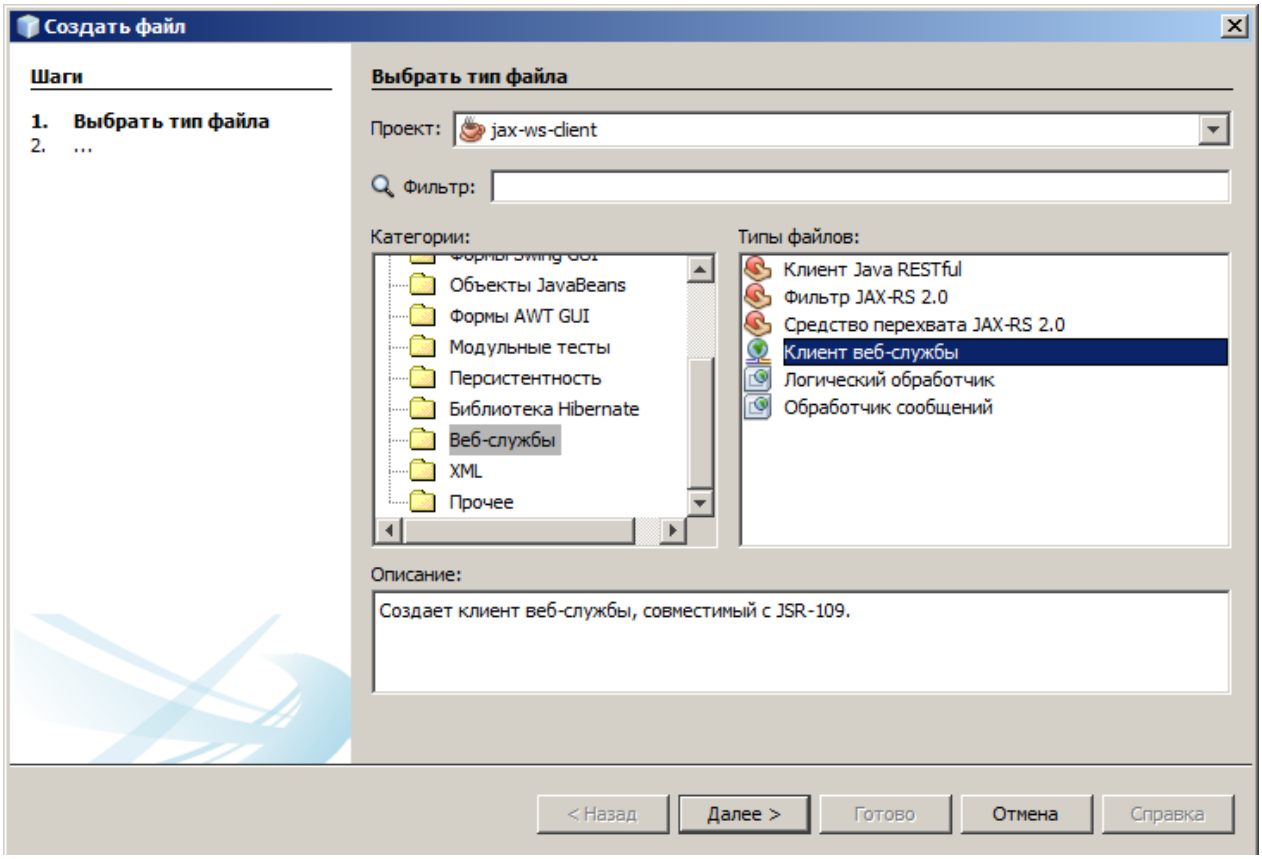
Зависимости JAX-RS клиента:

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.17.1</version>
</dependency>

<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-json</artifactId>
    <version>1.17.1</version>
</dependency>
```

Генерация артефактов веб-сервиса в IDE NetBeans

Для генерации артефактов веб-сервиса в IDE NetBeans необходимо кликнуть правой клавишей мыши по иконке проекта, в который необходимо добавить артефакты, выбрать “Новый” -> “Другое”. Затем в появившемся окне необходимо выбрать категорию “Веб-службы”, и выбрать тип файла “Клиент веб-службы”, как это показано на следующем скриншоте:



Настройка подключения к базе данных в GlassFish

Непосредственно перед созданием подключения к базе данных, необходимо в библиотеки домена GlassFish добавить JDBC-драйвер БД PostgreSQL. Для этого нужно jar-файл с JDBC-драйвером скопировать в директорию /glassfish4_dir/glassfish/lib.

После этого следует запустить сервер приложений, и, с помощью веб-браузера перейти в панель администрирования GlassFish (http://localhost:4848/, если не были переопределены настройки по умолчанию).

После логина на панели справа следует перейти в раздел Resources, затем JDBC->JDBC Connection Pools. Здесь необходимо создать новый пул подключений. После нажатия на кнопку “New”, в появившемся окне ввести любое значение в Pool Name и запомнить его, Resource Type=”javax.sql.DataSource”, Vendor=”Postgresql”. Далее нажать на кнопку “Next”. В появившемся окне следует перейти на вкладку Additional Parameters, или прокрутить страницу в самый низ. Здесь будет отображена таблица с параметрами для подключения. Из нее следует оставить только параметры с именами: portNumber, databaseName, serverName, user, password. Остальные параметры следует удалить. Значения данных параметров следует заполнить в соответствии с настройками СУБД. Далее представлена таблица, содержащая значения этих параметров для примера:

portNumber	5432
databaseName	ifmo-ws
serverName	Localhost
user	ifmo-ws
password	ifmo-ws

После того, как эти параметры заданы, следует нажать на кнопку “Save”. Далее перейти на вкладку “General” и нажать на кнопку “Ping”. В случае, если параметры заданы верно, и СУБД запущена, то будет выведено сообщение “Ping Succeeded”.

Далее на панели справа следует перейти в Resources->JDBC->JDBC Resources и нажать на “New”. В появившемся окне ввести JNDI Name=jdbc/ifmo-ws, а в выпадающем списке Pool Name выбрать созданный на предыдущем шаге пул. Нажать на “OK”. Теперь, если в таблице Resources вы видите ресурс с JNDI Name=jdbc/ifmo-ws, и с Connection Pool, соответствующем созданному пулу подключений, то настройка соединения с БД на GlassFish произведена успешно.

Ссылки и рекомендуемая литература

1. Машнин Т. С. Web-сервисы Java. – СПб.: БХВ-Петербург, 2012. – 560 с.
2. Mark D. Hansen. SOA Using Java Web Services. – NJ, 2007. – 574 p.
3. Leonard Richardson, Sam Ruby. RESTful Web Services. – O'Reilly media, 2007. – 454 p.
4. http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf
5. http://www.ibm.com/developerworks/ru/library/wa-jaxrs/
6. http://www.ibm.com/developerworks/webservices/library/ws-restful/
7. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
8. http://www.oracle.com/technetwork/testcontent/regrep-sales-overview-pub-128891.pdf
9. http://www.soa.com/products/registry
10. http://www.inst-informatica.pt/servicos/informacao-e-documentacao/biblioteca-digital/arquitectura-e-desenvolvimento-de-aplicacoes/soa/soa_registry_wp.pdf
11. http://www.mulesoft.com/resources/esb/service-registry-repository
12. http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/atc_comms_services/swim/documentation/media/briefings/Service_Registry_Brown_Bag_March2011_v0.5_030911.ppt
13.