Оглавление

Web Service Standard

Абстрактно веб-служба (Web Service, WS) – некая стандартная платформа, обеспечивающая возможность взаимодействия приложений по сети. WS получили распространение относительно недавно. Они дали реальные возможности в реализации кроссплатформенности (межоперабельности): взаимодействия приложений на различном железе, языках программирования, под управлением различных ОС. Это сделало WS промышленным стандартом связи A2A (приложение-приложение) и B2B (электронная коммерция). Для проектирования WS появилась сервис-ориентированная архитектура (Service-oriented Architecture, SOA), основной принцип которой заключается в экспорте функциональности приложений в виде слабосвязанных служб.

Rem: слабая связанность WS означает отделение клиента (он же потребитель, consumer) от реализации (поставщик, producer) при помощи обмена данными универсальным способом (XML, HTTP).

Наиболее популярные WS опираются на давно известные технологии XML/WSDL/SOAP и HTTP/REST. Это обеспечило их повсеместный выбор в области создания систем Enterprise-уровня такими вендорами как Oracle, IBM, Microsoft, JBoss, HP, BEA и пр.

Компания Sun Microsystems ввела WS в JEE. Точнее Sun и Java Community Process разработали API нескольких служб, в т.ч.:

- JAX-RS (Java API for RESTful, JSR 399)
- JAX-WS (Java API for XML WS, JSR 224)
- JAX-PRC (устаревший Java API for XML-based RPC)
- SAAJ (SOAP with Attachment API for Java)
- JAXR (Java API for XML Registries)

Эти API частично появились в J2EE 1.4 и были расширены в последующих версиях, включая JEE7. Фундаментальными технологиями для них являются:

- HTTP (RFC 2616)
- XML Schema
- XML namespace
- SOAP
- WSDL

Начиная с JEE7 поддерживается технология JSON (JavaScript Object Notation), которая может использоваться для создания компактных ответов веб-служб JAX-RS, особенно удобных для WEB 2.0 - клиентов, использующих JavaScript и технологию AJAX (Asynchronous Javascript and XML).

Наиболее важными из служб JEE являются JAX-RS и JAX-WS – дальнейшее развитие JAX-RPC.

Обзор Web-служб JEE

В JEE принято узкое определение WS: это удаленное приложение, либо описывающееся WSDL и вызывающееся через протокол SOAP в соответствии с правилами, описанными уточняющей спецификацией WS-I Basic Profile 1.1, либо это удаленное приложение, вызывающееся по HTTP-протоколу согласно правилам RESTful. В основе SOAP и WSDL лежат XML-технологии XML Schema и Xml namespace. В основе RESTful лежит прикладной протокол HTTP и абстрактно-логическая концепция передачи репрезентативного состояния (Representational State Transfer, REST).

Функционально WS представляет собой набор операций, которые можно вызывать удаленно. Аналогом операций в Java являются методы, которые принимают примитивы или объекты и выдают в ответ также примитивы или объекты. Однако идеология операций WS несколько уже, они должны выполнять некоторую законченную работу вроде получения списка товаров, размещения заказа и т.п., их не используют для реализации аналогов get/set методов и прочих вспомогательных действий.

Отличие WS от технологии удаленного доступа Java RMI (с мультиязычным протоколом RMI-IIOP) состоит в отсутствии поддержки состояния: в RMI используются прокси-объекты, являющиеся представлениями удаленных объектов, реализующие как поддержку их состояния, так и вызовы их удаленных методов; WS не представляется объектом и не поддерживает сессии, все запросы любых операций рассматриваются как взаимонезависимые.

Rem: взаимонезависимость тут подразумевает неизменность бизнес-логики самой службы (объектов ее реализации), при этом состояние бизнес-модели (БД, файлы, etc) может меняться от запроса к запросу и результаты таких запросов будут строго говоря зависимыми. В концепции REST вводятся уточняющие понятия, относящиеся к независимости последовательных запросов: идемпотентность (AA=A) и безопасность (Ax=x), где A – оператор, описывающий запрос, а x – состояние бизнес-модели.

В отличие от асинхронного обмена сообщениями MDB, вызовы JAX-WS выполняются синхронно, т.е. клиент ждет ответа на запрос к WS (если операция должна выдавать ответ).

Наиболее близкими по поведению к WS являются SLSB. Поэтому они обычно и используются при экспорте из EJB в виде WS. Также допускается создание WS на основе @Singleton EJB и POJO.

SOAP WS

Веб-служба SOAP целиком построена на технологии XML. Формат сообщений, которыми обмениваются потребитель и поставщик, протокол обмена (SOAP) и описание самой веб-службы (WSDL) строятся на XML. Кроме этого, для обеспечения автоматической проверки сообщений и предотвращения конфликта имен используются спецификация XML Schema и стандарт XML Namespace.

XML - схема (XML Schema)

XML (eXtended Markup Language) – расширяемый язык разметки, по сути язык для написания целевых разметочных языков. XML – схема, как и ее функциональный аналог DTD, позволяет проверить XML-документ на корректность. Корректность измеряется по 2 критериям, корректный документ должен быть:

- well formed – правильно сформированным
- valid – допустимым

Чтобы быть правильно-сформированным, XML-документ должен подчиняться синтаксическим правилам языка разметки XML: использованию правильных символов для указания начала и конца элементов, правильному определению атрибутов и т.д. Большинство парсеров, основанных на стандартах SAX и DOM автоматически проверяют XML-документы на правильность формы.

В дополнению к правильному оформлению, XML-документ должен быть допустимым, т.е. использовать только определенные типы элементов в правильном порядке. Критерии валидации не могут определяться только синтаксическими правилами XML. Это в большей мере прерогатива приложения, в котором используется XML-документ. Например, XML-документ может быть недопустимым, если не содержит элементов почтового адреса или телефона. Для валидации документов надо предоставить способ обеспечения прикладных ограничений.

Ex: пример XML-схемы (XSD) адреса

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:titan="http://www.titan.com/Reservation"
 targetNamespace="http://www.titan.com/Reservation">
 <element name="address" type="titan:AddressType"/>
 <complexType name="AddressType">
  <sequence>
   <element name="street" type="string"/>
   <element name="city" type="string"/>
   <element name="state" type="string"/>
   <element name="zip" type="string"/>
  </sequence>
 </complexType>
</schema>
```

В примере элемент <complexType> определяет тип элемента, подобно тому, как в Java класс объявляет тип Java-объектов. В <complexType> определены имена, типы и порядок элементов: элемент AddressType может содержать 4 строковых элемента street, city, state, zip. Валидация в данном случае довольно строгая — любой XML-документ, удовлетворяющий данной XML-схеме должен содержать указанные типы в строгом порядке:

Ex: пример XML-схемы адреса

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<address>
 <street>800 Langdon Street</street>
 <city>Madison</city>
 <state>WI</state>
```

```
  <zip>53706</zip>
</address>
```

XML-схема (XSD) автоматически поддерживает пару дюжин типов данных, называемых встроенными типами XSD и являющихся ее частью. Эти типы поддерживаются любым XML-совместимым парсером.

Встроенный тип XSD	Аналогичные типы Java
byte	Byte, byte
boolean	Boolean, boolean
short	Short, short
int	Integer, int
long	Long, long
float	Float, float
double	Double, double
string	java.lang.String
dateTime	java.util.Calendar
integer	java.math.BigInteger
decimal	java.math.BigDecimal

По-умолчанию любой элемент, определенный в <complexType>, может встречаться в XML-документе ровно 1 раз. Но можно переопределить элемент как необязательный либо как многократный путём исправления атрибута частоты появления (occurence):

```
<element … minOccurs=”1” maxOccurs=”2” ...>
```

В данном примере элемент должен присутствовать в XML-документе в количестве 1 до 2 экземпляров. По-умолчанию minOccurs=maxOccurs=”1”. Допустимы значения minOccurs=”0” (необязателен) и maxOccurs=”unbounded” (неограниченное число).

Порядок следования элементов задается элементами <sequence> и <all>. Элемент <sequence> требует расположения элементов в XML-документе в том же порядке, что в схеме. Элемент <all> допускает произвольное расположение.

Помимо объявления в схеме встроенных типов XML, можно использовать элементы <complexType>. Это аналогично объявлению полей ссылочных типов в Java.

Ex: пример XSD для XML документов, описывающих заказчика

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:titan="http://www.titan.com/Reservation"
```

```
  targetNamespace="http://www.titan.com/Reservation">
 <element name="customer" type="titan:CustomerType"/>
 <complexType name="CustomerType">
  <sequence>
   <element name="last-name" type="string"/>
   <element name="first-name" type="string"/>
   <element name="address" type="titan:AddressType"/>
  </sequence>
 </complexType>
 <complexType name="AddressType">
  <sequence>
   <element name="street" type="string" />
   <element name="city" type="string"/>
   <element name="state" type="string"/>
   <element name="zip" type="string"/>
  </sequence>
 </complexType>
</schema>
```

В данной XSD сказано, что элемент типа CustomerType должен содержать последовательно строковые элементы <last-name>, <fist-name> и элемент <address> типа AddressType. Перед типом AddressType для обозначения его пространства имен использован префикс titan: .

Ex: пример XML-документа, описывающего заказчика согласно предыдущей XSD

```
<?xml version='1.0' encoding='UTF-8' ?>
<customer>
 <last-name>Jones</last-name>
 <first-name>Sara</first-name>
 <address>
  <street>3243 West 1st Ave.</street>
  <city>Madison</city>
  <state>WI</state>
  <zip>53591</zip>
 </address>
</customer>
```

Rem: в качестве типа можно использовать абстрактный XSD-тип anyType, также есть возможность задавать абстрактный множественный тип при помощи <xsd:choice>.Валидация их пройдет без проблем, но при парсинге будут сложности.

Можно составлять более сложные схемы.

Ex : пример XSD бронирования, использующей несколько связанных сложных типов

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:titan="http://www.titan.com/Reservation"
   targetNamespace="http://www.titan.com/Reservation">
```

```
 <element name="reservation" type="titan:ReservationType"/>
 <complexType name="ReservationType">
  <sequence>
   <element name="customer" type="titan:CustomerType"/>
   <element name="cruise-id" type="int"/>
   <element name="cabin-id" type="int"/>
   <element name="price-paid" type="double"/>
  </sequence>
 </complexType>
 <complexType name="CustomerType">
  <sequence>
   <element name="last-name" type="string"/>
   <element name="first-name" type="string"/>
   <element name="address" type="titan:AddressType"/>
   <element name="credit-card" type="titan:CreditCardType"/>
  </sequence>
 </complexType>
 <complexType name="CreditCardType">
  <sequence>
   <element name="exp-date" type="dateTime"/>
   <element name="number" type="string"/>
   <element name="name" type="string"/>
   <element name="organization" type="string"/>
  </sequence>
 </complexType>
 <complexType name="AddressType">
  <sequence>
   <element name="street" type="string"/>
   <element name="city" type="string"/>
   <element name="state" type="string"/>
   <element name="zip" type="string"/>
  </sequence>
 </complexType>
</schema>
```

XML-документ, который согласуется со схемой резервации должен содержать информацию о заказчике (имя и адрес), информацию о кредитке, код круиза и каюты, которая резервируется:

Ex: пример XML-документа, описывающего бронирование согласно предыдущей XSD

```
<?xml version='1.0' encoding='UTF-8' ?>
<reservation>
 <customer>
  <last-name>Jones</last-name>
  <first-name>Sara</first-name>
  <address>
   <street>3243 West 1st Ave.</street>
   <city>Madison</city>
   <state>WI</state>
   <zip>53591</zip>
```

```
  </address>
  <credit-card>
   <exp-date>09-2007/exp-date>
   <number>0394029302894028930</number>
   <name>Sara Jones</name>
   <organization>VISA</organization>
  </credit-card>
 </customer>
 <cruise-id>123</cruise-id>
 <cabin-id>333</cabin-id>
 <price-paid>6234.55</price-paid>
</reservation>
```

Этот XML-документ может быть послан в качестве запроса гипотетическим турагентством гипотетическому туроператору, если по каким-то причинам нельзя запросить напрямую гипотетический туроператорский TravelAgentEJB.

В run-time XML-валидатор сравнит XML-документ с его схемой, проверяя выполнение ее правил. Если проверка (валидация) покажет несоответствие, то XML-документ будет признан недопустимым, а парсер выбросит исключение или сообщение об ошибке. Т.е. по XSD проверяется, что полученный приложением XML-документ является правильно-структурированным. Это дает гарантию, что при парсинге и извлечении информации не возникнет ошибки. Если же, например, в <reservation> предыдущего примера пропущен обязательный согласно схеме элемент <credit-card>, то XSD-валидация заранее выдаст ошибку и она не возникнет в коде из-за отсутствия этого ключевого для бронирования элемента.

XML - пространство имен (XML Namespaces)

XSD определяет язык разметки для XML-документов определенного вида. Также как java-класс определяет тип java-объекта, XSD определяет тип XML-документа. Допускается сочетание множества языков разметки в едином документе так, что элементы из этих различных языков будут проходить валидацию отдельно по своим XSD.

Для использования разных языков разметки в едином документе надо строго определить, какой элемент относится к какому языку.

Ex : использование 2 языков разметки в одном XML-документе (без пересечения)

```
<?xml version='1.0' encoding='UTF-8' ?>
<res:reservation xmlns:res="http://www.titan.com/Reservation">
 <res:customer>
  <res:last-name>Jones</res:last-name>
  <res:first-name>Sara</res:first-name>
  <addr:address xmlns:addr="http://www.titan.com/Address">
   <addr:street>3243 West 1st Ave.</addr:street>
```

```
    <addr:city>Madison</addr:city>
    <addr:state>WI</addr:state>
    <addr:zip>53591</addr:zip>
   </addr:address>
   <res:credit-card>
    <res:exp-date>09-2007</res:exp-date>
    <res:number>0394029302894028930</res:number>
    <res:name>Sara Jones</res:name>
    <res:organization>VISA</res:organization>
   </res:credit-card>
  </res:customer>
  <res:cruise-id>123</res:cruise-id>
  <res:cabin-id>333</res:cabin-id>
  <res:price-paid>6234.55</res:price-paid>
</res:reservation>
```

Элементы каждого языка разметки могут снабжаться префиксами вида xxx: , где xxx – имя префикса, относящееся к одному из языков разметки. Префиксы вводятся при помощи объявления пространства имен XML в каком-то из элементов XML-документа, которое действительно для всех его потомков.

Пространство имен XML — уникальный идентификатор конкретного языка XML-разметки, определенного в соответствующем XSD. Состоит из префикса (короткого локального имени, уникального для использующего его XML-документа) и URI – идентификатора, уникального для всей области использования этого языка разметки.

Это объявление имеет вид:

```
xmlns:prefix = “URI”
```

Атрибут xmlns – объявление пространства имен (XML Namespace).

Префикс prefix — краткое имя (алиас) языка разметки, уникальное для использующего его XML-документа. Это любое незанятое имя без пробелов и спецсимволов, негласно принято обозначать префиксы в виде сокращения от соответствующего им языка разметки.

Значение URI – уникальный идентификатор языка разметки. Он имеет формат унифицированного индикатора ресурса (Uniform Resource Identifier). В отличие от префикса, значение URI должно строго следовать определенному формату. Часто в качестве URI используют его общеупотребимый частный случай — URL.

Rem: URL, используемый в качестве идентификатора языка разметки, не несет никакой смысловой нагрузки (адреса WWW, указателя на файл и т.п.). Единственное требование к нему — уникальность в области использования соответствующего языка разметки.

```
Ex: примеры объявления адресных пространств
```

```
xmlns:addr="http://www.titan.com/Address"
xmlns:res="http://www.titan.com/Reservation"
```

XML-парсер будет сопоставлять URI из объявления пространства имен в XML-документе с целевым URI, описанном в атрибуте targetNamespace XSD:

Ex : XSD с объявлением целевого пространства имен

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.titan.com/Address">
 <complexType name="AddressType">
   <sequence>
     <element name="street" type="string"/>
     <element name="city" type="string"/>
     <element name="state" type="string"/>
     <element name="zip" type="string"/>
   </sequence>
 </complexType>
</schema>
```

Атрибут targetNamespace определяет уникальный целевой URI языка разметки, описываемого в XSD. Он является постоянным идентификатором своей XSD. И всякий раз, когда элементы из XSD-схемы используются в XML-документе, XML-документ должен использовать объявление пространства имен для определения принадлежности этих элементов соответствующему языку разметки.

Пространство имен, определенное в корне поддерева элементов, распространяется на все это поддерево. Поэтому объявление достаточно делать один раз и далее можно просто помечать нужные элементы соответствующими префиксами.

Однако приписывание к каждому элементу XML-документа префикса утомляет. Поэтому в XML-документ введено пространство имен по-умолчанию. Его объявление имеет вид: xmlns=”URI”. Пример выше XML-документа с явным пространством имен и префиксами можно переписать:

Ex : XML-документ с объявлением пространства имен по-умолчанию (без префиксов)

```
<?xml version='1.0' encoding='UTF-8' ?>
<reservation xmlns="http://www.titan.com/Reservation">
 <customer>
   <last-name>Jones</last-name>
   <first-name>Sara</first-name>
   <addr:address xmlns:addr="http://www.titan.com/Address">
     <addr:street>3243 West 1st Ave.</addr:street>
     <addr:city>Madison</addr:city>
     <addr:state>WI</addr:state>
```

```
    <addr:zip>53591</addr:zip>
   </addr:address>
   <credit-card>
    <exp-date>09-2007</exp-date>
    <number>0394029302894028930</number>
    <name>Sara Jones</name>
    <organization>VISA</organization>
   </credit-card>
  </customer>
  <cruise-id>123</cruise-id>
  <cabin-id>333</cabin-id>
  <price-paid>6234.55</price-paid>
</reservation>
```

Любой элемент без префикса считается принадлежащим пространству имен по-умолчанию. В вышеприведенном примере это пространство имен «http://www.titan.com/Reservation» (язык разметки для бронирования). Пространство имен по-умолчанию имеет границы или область действия. Область действия начинается с объявления в одном из элементов пространства имен по-умолчанию и распространяется на всех потомков этого элемента без префиксов. В одно пространство имен по-умолчанию можно вложить другое пространство имен по-умолчанию.

Ex : XML-документ с объявлением вложенных пространств имен по-умолчанию

```
<?xml version='1.0' encoding='UTF-8' ?>
<reservation xmlns="http://www.titan.com/Reservation">
  <customer>
    <last-name>Jones</last-name>
    <first-name>Sara</first-name>
    <address xmlns="http://www.titan.com/Address">
      <street>3243 West 1st Ave.</street>
      <city>Madison</city>
      <state>WI</state>
      <zip>53591</zip>
    </address>
    <credit-card>
      <exp-date>09-2007</exp-date>
      <number>0394029302894028930</number>
      <name>Sara Jones</name>
      <organization>VISA</organization>
    </credit-card>
  </customer>
  <cruise-id>123</cruise-id>
  <cabin-id>333</cabin-id>
  <price-paid>6234.55</price-paid>
</reservation>
```

При этом вложенное пространство имен по-умолчанию будет перекрывать в своем поддереве внешнее пространство имен по-умолчанию. Пространство имен по-

умолчанию не действует на беспрефиксные атрибуты; принадлежность беспрефиксных атрибутов к пространству имен определяется только элементом, в котором они расположены. Исключением являются служебные атрибуты XML вроде xmlns, являющиеся частью языка XML.

Rem: префиксы для атрибутов используются при смешении языков разметки. Пример — применение языка связывания xlink .

URI из объявления пространства имен не несет смысловой нагрузки указателя местоположения схемы, это просто уникальный идентификатор. Для задания местоположения XSD, связанного с пространством имен, в XML-документе можно использовать атрибут schemaLocation=”xxx yyy”, где xxx – URI пространства имен, а yyy – URL расположения XSD, по которому эту схему может скачать XML-парсер.

Ex : задание местоположения XSD в атрибуте schemaLocation XML-документа

```
<?xml version='1.0' encoding='UTF-8' ?>
  <reservation xmlns="http://www.titan.com/Reservation"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
    xsi:schemaLocation="http://www.titan.com/Reservation
      http://www.titan.com/schemas/reservation.xsd">
...
```

В вышеприведенном фрагменте XML-документа, в элементе <reservation> определено пространство имен по-умолчанию (беспрефиксное) с URI “http://www.titan.com/Reservation”, а URL реального местоположения схемы “http://www.titan.com/cshemas/reservation.xsd” связан с этим пространством имен при помощи атрибута xsi:schemaLocation.

Атрибут schemaLocation не является частью языка XML, этот атрибут определен в схеме XSI с URI ”http://www.w3.org/2001/XMLSchema-Instance”. Поэтому для использования schemaLocation надо сначала определить пространство имен схемы XSI и использовать префикс, в качестве которого обычно выбирают “xsi”.

XSD является XML-документом, в нем точно также можно использовать пространства имен. Основная схема для XSD — XML Schema (URI – “http://www.w3.org/2001/XMLSchema”). Часто ее объявляют в XSD пространством имен по-умолчанию, либо используют префикс (обычно это “xs” или “xsd”). В частности, в этой схеме определены элемент <schema> и его атрибут targetNamespace, задающий URI XSD . В XSD-документе также могут использоваться и другие схемы со своими пространствами имен.

Ex : пример XSD адреса, рассматриваемого как XML-документ

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema
```

```
   xmlns=http://www.w3.org/2001/XMLSchema
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance
   xmlns:addr=http://www.titan.com/Address
   xmlns:res=http://www.titan.com/Reservation
   targetNamespace="http://www.titan.com/Reservation">
  <import namespace="http://www.titan.com/Address"
            xsi:schemaLocation="http://www.titan.com/Address.xsd" />
  <element name="reservation" type="res:ReservationType"/>
  <complexType name="AddressType">
   <sequence>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
    <element name="state" type="string"/>
    <element name="zip" type="string"/>
   </sequence>
  </complexType>
</schema>
```

В данном фрагменте XSD выражение xmlns="http://www.w3.org/2001/XMLSchema" объявляет пространством имен по-умолчанию основную XML Schema, что избавляет от префиксов большинство элементов. Атрибут targetNamespace="http://www.titan.com/Address" задает URI целевого пространства имен, к которому будут принадлежать все объявленные в данном XSD элементы и которое будет использоваться в XML-документах, использующих этот язык разметки. Выражение xmlns:addr="http://www.titan.com/Address" задает пространство имен с префиксом ”addr”, совпадающее с целевым, т.е. собственное пространство имен данного XSD, что может потребоваться для определения в XSD типизированных элементов. Элемент <element name="address" type="addr:AddressType"/> задает типизированный элемент <address>, тип задается F.Q.N. (QName): “префикс:тип_элемента”. По аналогии с Java: в классе Address задается поле address типа вложенного класса AddressType.

В XSD-документе может определяться множество типов, элементов и схем языков разметки, допускается их смешивание.

Ex : пример XSD бронирования, рассматриваемого как XML-документ

```
<?xml version='1.0' encoding='UTF-8' ?>
<schema
   xmlns=http://www.w3.org/2001/XMLSchema
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance
   xmlns:addr=http://www.titan.com/Address
   xmlns:res=http://www.titan.com/Reservation
   targetNamespace="http://www.titan.com/Reservation">
  <import namespace="http://www.titan.com/Address"
            xsi:schemaLocation="http://www.titan.com/Address.xsd" />
  <element name="reservation" type="res:ReservationType"/>
  <complexType name="ReservationType">
   <sequence>
```

```
    <element name="customer" type="res:CustomerType "/>
    <element name="cruise-id" type="int"/>
    <element name="cabin-id" type="int"/>
    <element name="price-paid" type="double"/>
   </sequence>
  </complexType>
  <complexType name="CustomerType">
   <sequence>
    <element name="last-name" type="string"/>
    <element name="first-name" type="string"/>
    <element name="address" type="addr:AddressType"/>
    <element name="credit-card" type="res:CreditCardType"/>
   </sequence>
  </complexType>
  <complexType name="CreditCardType">
   <sequence>
    <element name="exp-date" type="dateTime"/>
    <element name="number" type="string"/>
    <element name="name" type="string"/>
    <element name="organization" type="string"/>
   </sequence>
  </complexType>
</schema>
```

В данном XSD XS-элемент <import namespace="http://www.titan.com/Address"
xsi:schemaLocation="http://www.titan.com/Address.xsd" />
подтягивает чужую схему, определяя в XS-атрибуте namespace ее URI, а в XSI -атрибуте schemaLocation URL фактического местонахождения. Типом элемента может быть любой из встроенных XML-типов, которые определены в XS, простой и сложные типы. В примере XML-типы идут без префикса, т.к. XS объявлена в пространстве имен по-умолчанию, но часто в XSD задают и используют для XS префиксы "xs" или "xsd".

SOAP

SOAP – просто распределенный объектный протокол наподобие DCOM или CORBA IIOP.

Языки разметки SOAP основаны на своих собственных XSD и в работе полагаются на пространства имен XML. Каждое сообщение SOAP представляет собою сочетание SOAP-элементов и прикладной информации. Стандартные элементы SOAP можно отличить по соответствующему пространству имен.

Ex : пример SOAP-сообщения формата Document/Literal

```
<?xml version='1.0' encoding='UTF-8' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
  <env:Body>
```

```
    <reservation xmlns="http://www.titan.com/Reservation">
        <customer>
            <!-- customer info goes here -->
        </customer>
        <cruise-id>123</cruise-id>
        <cabin-id>333</cabin-id>
        <price-paid>6234.55</price-paid>
    </reservation>
  </env:Body>
</env:Envelope>
```

Главной задачей SOAP является создание основанного на XML фрэймворка для упаковки прикладной информации, передаваемой между различными программными платформами (Java, Perl, .NET, PHP и т.д.).

Для реализации этой цели в SOAP объявлено несколько типов элементов. Ниже структура SOAP:

Элемент	**Функциональное назначение**
Envelope	определяет само сообщение и пространства его имен
Header	техническая информация обмена, опционально прикладные сообщения
Body	полезная нагрузка, т.е. сообщение обмена
Fault	необязательная информация об ошибках

Корневым элементом является <Envelope> (обертка), все остальные элементы сообщения содержатся в нем. У <Envelope> два непосредственных потомка <Header> и <Body>.

Элемент <Header> необязателен в SOAP, там хранится вспомогательная инфраструктурная информация: ID транзакций, маршрутные данные, контекст безопасности и т.д. В вышеприведенном примере элемент <Header > пуст, что характерно для основных WS, т.к. в большинстве случаев требуется лишь обмен прикладной информацией, а не более навороченные варианты, связанные контекстом транзакций или безопасности, например.

Элемент <Body> обязательный для SOAP. Он содержит прикладную информацию. Профиль WS-I требует наличия в <Body> единственного прямого потомка, т.е. прикладная XML-структура должна быть деревом. В примере выше <Body> содержит XML-фрагмент, определенный схемой Reservation и являющийся деревом.

Элемент <Fault> необязателен, он представляет в ответах информацию об исключительных ситуациях, возникающих при сбоях во время обработки запросов или передаче.

Rem: вложенные в SOAP XML-документы называют фрагментами.

Стили SOAP-сообщений

Сообщение SOAP является Document/Literal сообщением, если содержимое <Body> представляет собой цельный XML-документ, что подразумевает возможность валидации всего сообщения. Это предпочтительный вид SOAP-сообщений. При проектировании сообщений стиля Document/Literal применяются шаблоны Bare и Wrapped. В шаблоне Bare элементы всех параметров операции описываются в единой схеме, но т.к. WS-I Basic Profile 1.1 требует единственного потомка <Body> SOAP-сообщения, то для случая множества параметров требуется либо выносить параметры в SOAP-элемент <Header> , либо определять в этой схеме элемент-обертку. Последнее может потребовать переделки сигнатуры операции/метода. Шаблон Wrapped предписывает описывать сообщение в схеме деревом, имя корня-обертки которого должно совпадать с наименованием соответствующей операции, что позволяет автоматизировать процесс обертывания/развертывания и не затрагивать сигнатуру операции/метода.

В примере выше как раз Document/Literal SOAP-сообщение. XSI-атрибут schemaLocation для <reservation> пропущен, т.к. предполагается наличие схемы у принимающей стороны.

Другой тип сообщений, допускаемый WS-I Basic Profile 1.1 и EJB 3.1 – RPC/Literal. Эти сообщения описывают вызовы методов с параметрами и возвращаемым значением, каждое из которых задается собственной независимой XSD. В итоге PRC/Literal сообщение не обязано являться корректным XML-документом, не требует общего XSD и XML-валидации, а требует только для каждого параметра определить элемент с тем же именем и корректным типом и обернуть все это тегом с именем операции (последнее хорошо автоматизируется при автогенерации WSDL-документа). Такие сообщения быстрее обрабатываются, их легче спроектировать, но за это приходится платить пропуском XML-валидации и, следовательно, надежностью.

Рассмотрим пример описания вызова Java-метода SOAP-сообщением в RPC/Literal стиле.

Ex : пример Java-интерфейса с методом, который будет вызываться через SOAP

```
public interface TravelAgent {
   public void makeReservation(int cruiseId, int cabinId,
                               int customerId, double price);
}
```

Ex : пример SOAP-сообщения формата RPC/Literal, описывающего вызов метода makeReservation(…)

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

```
   xmlns:titan="http://www.titan.com/TravelAgent/">
  <env:Body>
    <titan:makeReservation>
        <cruiseId>23</cruiseId>
        <cabinId>144</cabinId>
        <customerId>9393</customerId>
        <price>5677.88</price>
    </titan:makeReservation>
  </env:Body>
</env:Envelope
```

Первый элемент <Body> определяет WS-операцию, которая будет вызываться. В нем определяются элементы параметров, каждый со своим значением.

Rem: ни веб-службы SOAP в JEE7 (JAX-WS 2.2A), ни WS-I Basic Profile 1.1 не поддерживают формат RPC/Encoded. В JEE6 веб-службы SOAP описывалась как спекой JAX-WS, так и спекой JAX-RPC, которая использовала RPC/Encoded. На заре WS он был употребим, но затем индустрия перешла на Document/Literal и Document/RPC типы сообщений, т.к. межоперационное взаимодействие между платформами, использующими RPC/Encoded было далеко от идеала и довольно сложно. Кроме того, RPC/Encoded опирался на SOAP-типы (массивы, списки, перечисления и т.п.), в отличие от Document/Literal и RPC/Literal, использующих только встроенные XML-типы, что выглядело менее предпочтительно для обеспечения кроссплатформенности и языконезависимости. Так что несмотря на поддержку JEE6 (EJB 3.1), использование RPC/Encoded не рекомендуется.

Обмен SOAP-сообщениями через HTTP

SOAP является безразличным к сетевому протоколу. Это означает, что SOAP независим от типа сети или протокола, использующегося в ней. Но вместе с этим SOAP использует при обмене в основном протокол HTTP. Использование этого прикладного протокола в качестве транспортного вызвано повсеместным распространением использования HTTP для Internet-продуктов: web-серверов, серверов приложений, сетевых устройств и т.п. Это повсеместное распространение HTTP сразу же обеспечивает инфраструктурой обмен сообщений SOAP (стандартный порт 80, обработчики). Это стало одним из факторов популярности SOAP.

Другим преимуществом использования транспорта HTTP для SOAP стала способность SOAP-сообщений проникать через фаерволы. При использовании абонентами файерволов передача данных по протоколам, отличным от HTTP и SMTP часто невозможна или затруднена. Использование HTTP решает эту проблему, облегчает жизнь разработчику и усложняет ее администраторам безопасности. Поддержка HTTP протоколов вроде SOAP обычно называется HTTP-туннелированием. Ранее другие распределенные объектные протоколы вроде CORBA UIIOP или DCOM использовали HTTP-туннелирование в единичных случаях проприетарно, что делало обеспечение межоперабельности весьма проблематичным. В спецификациях SOAP с версии 1.1 HTTP-туннелирование встроено, что гарантирует межоперабельность. Благодаря тому, что почти все вендоры серверов приложение реализуют SOAP, HTTP-туннелирование

стало повсеместным.

Rem: в спецификации SOAP 1.1 использовались лишь POST-запросы HTTP, в SOAP 1.2 появилась возможность использовать и GET-запросы со специальным MIME-типом application/soap+xml в заголовке Accept.

Можно использовать протокол SOAP 1.2, используя в качестве транспорта другие прикладные протоколы — SMTP, FTP или даже канонический транспортный TCP/IP. Но строго описано только HTTP-туннелирование. Поэтому в спецификации EJB 3.2 (JEE7) требуется только поддержка SOAP 1.2 над HTTP 1.1.

Посмотрел — не делай

На практике SOAP напрямую используется редко. Как и большинство протоколов, SOAP разработан для генерации и потребления через ПО и обычно инкапсулируется разработчиками API.

В EJB 3.1 API для обмена сообщениями SOAP называется JAX-WS. Он скрывает детали SOAP-обмена и позволяет разработчику сосредоточиться на разработке и использовании WS. При использовании JAX-WS иметь дело с SOAP придется крайне редко.

Rem: на самом деле иногда приходится снифферить невалидные SOAP-сообщения при анализе ошибок.

WSDL

Язык описания веб-сервисов WSDL очерчивает структуру SOAP WS и дает возможность ее полного описания.

WSDL является разметочным языком XML. Он независим от платформы, языка программирования и протокола. Последнее означает, что WSDL может работать с протоколами, отличными от SOAP и HTTP. Это делает WSDL гибким, но за это надо платить сильной абстрактностью и сложностью.

Поскольку WS-I Basic Profile 1.1 поддерживает только SOAP 1.1/1.2 поверх HTTP, можно ограничиться изучением WSDL, связанным только с этим протоколом.

Рассмотрим пример описания Java-интерфейса TravelAgent.

Ex : пример Java-интерфейса для обертывания

```
public interface TravelAgent {
   public void makeReservation(int cruiseID, int cabinID,
                               int customerId, double price);
}
```

Цель в том, чтобы любое приложение могло вызвать этот метод используя SOAP, невзирая на язык программирования, на котором он реализован и платформу, на которой он запущен.

Поскольку потенциальные клиенты — приложения, которые могут не понимать Java, необходимо создать обертку вокруг метода на понятном им языке XML, иными словами преобразовать Java-метод в операцию WS. Используя XML, точнее язык разметки WSDL, можно описать сообщения, которые надо послать для вызова операции WS, дергающей метод.

Ex : пример WSDL-описания WS, обертывающего интерфейс TravelAgent

```
<?xml version="1.0"?>
<definitions name="TravelAgent"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:titan="http://www.titan.com/TravelAgent"
   targetNamespace="http://www.titan.com/TravelAgent">
<!-- элементы сообщения, описывающие параметры и возвращаемые значения -->
<message name="RequestMessage">
   <part name="cruiseId"   type="xsd:int" />
   <part name="cabinId"    type="xsd:int" />
   <part name="customerId" type="xsd:int" />
   <part name="price"      type="xsd:double" />
</message>
<message name="ResponseMessage">
  <part name="reservationId" type="xsd:string" />
</message>
<!-- элемет portType, описывающий абстрактный интерфейс WS -->
<portType name="TravelAgent">
  <operation name="makeReservation">
     <input message="titan:RequestMessage"/>
     <output message="titan:ResponseMessage"/>
  </operation>
</portType>
<!-- связывающий элемент, указывающий какие используются протокол и кодировка-->
<binding name="TravelAgentBinding" type="titan:TravelAgent">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="makeReservation">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
      </input>
      <output>
        <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
      </output>
   </operation>
</binding>
```

```
<!-- элемент службы, указывающий URL-адрес WS -->
<service name="TravelAgentService">
  <port name="TravelAgentPort" binding="titan:TravelAgentBinding">
    <soap:address location="http://www.titan.com/webservices/TravelAgent" />
  </port>
</service>
</definitions>
```

На первый взгляд пример непонятен, поэтому будет изучен по частям. Следует заметить, что большинство платформ, поддерживающих WS, обладают инструментами для генерации WSDL. Поэтому до WSDL дело обычно доходит только тогда, когда надо взглянуть на причину сбоя описанного им WS. В любом случае познакомиться с WSDL полезно, даже если вручную писать на нем ничего не придется.

Структура WSDL-документа

Корневой элемент <definitions> определяет пространства имен всего WSDL-документа. В элементе <types> определяются вложенными схемами (XSD) типы данных, составляющие сообщения. В разделе сообщений (элементы <message>) определяются сообщения, которыми идет обмен со службой, части сообщений представляют входные и выходные данные операций. В разделе интерфейсов (элементы <portType> - они же порты) определены доступные операции, каждая из которых ссылается на входное и выходное сообщения. Элемент связывания <binding> описывает протокол обмена для каждого порта, описывает формат сообщений для каждой операции порта. Элемент <service> определяет для связок портов конечные точки (Endpoint WS) в формате URL.

Элемент <definitions>

Элемент <definitions> - корневой элемент WSDL. Обычно тут указываются все используемые пространства имен. В примере выше там объявлены несколько пространств имен:

```
Ex: пространства имен корня WSDL
```

```
<definitions name="TravelAgent"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:titan="http://www.titan.com/TravelAgent"
  targetNamespace="http://www.titan.com/TravelAgent">
```

- пространство имен по-умолчанию xmlns="http://schemas.xmlsoap.org/wsdl/" схемы WSDL;
- для XS выбрано пространство имен xmlns:xsd="http://www.w3.org/2001/XMLSchema", оно используется для определения типов вроде type=”xsd:int”;
- targetNamespace="http://www.titan.com/TravelAgent" задает целевое пространство имен текущего WSDL-документа, которое имеет сходное назначение с XSI-атрибутом targetNamespace, используемым в XSD, но используется для привязки элементов WSDL не по их именам, а по их атрибутам name;
- пространство имен xmlns:titan=”http://www.titan.com/TravelAgent” совпадает с целевым пространством имен WSDL-документа задавая ему тем самым префикс "titan". Например, ”titan:xxx” означает связь с элементом WSDL-документа <yyy … name=”xxx”>.

Элементы <portType> и <message>

Эти элементы являются непосредственными потомками корневого <definitions>.

Параметры и возвращаемые значения операций описываются в дочерних элементах <part> элемента сообщения <message>.

Описание параметра зависит от формата сообщения. Для формата RPC/Literal используется атрибут ”type”, значением которого должен быть XML-тип данного сообщения — встроенный XML-тип или прикладной тип, заданный вложенной схемой в элементе <types>:

```
<!-- пример описания параметров и результата для формата RPC/Literal -->

<message name="RequestMessage">
  <part name="cruiseId" type="xsd:int" />
  <part name="cabinId" type="xsd:int" />
  <part name="customerId" type="xsd:int" />
  <part name="price" type="xsd:double" />
</message>
<message name="ResponseMessage">
 <part name="reservationId" type="xsd:string" />
</message>
```

Стиль Document/Literal требует определения атрибута “element”, значением которого должен быть определенный в элементе <types> простой или сложный XML-элемент:

```
...  <part name="cruiseId" element="cruiseIdElement" /> ...
```

Каждый элемент <portType> задает тип порта (интерфейса) WS, описывая набор операций WS. Описанный в примере <portType name="TravelAgent"> соответствует методам Java-интерфейса TravelAgent:

```
<!-- элемет portType, описывающий абстрактный интерфейс WS -->
<portType name="TravelAgent">
  <operation name="makeReservation">
    <input message="titan:RequestMessage"/>
    <output message="titan:ResponseMessage"/>
  </operation>
</portType>
```

<operation> описывает отдельную операцию WS, может содержать элементы <input>, <output> и <fault>. Элемент <input> описывает тип SOAP сообщения, который клиент будет посылать WS. Элемент <output> описывает тип SOAP сообщения, который клиент ожидает увидеть в ответ. Элемент <fault> описывает тип SOAP сообщения об ошибке, которое может быть послано клиенту WS (аналог Java Exception).

JAX-WS и, следовательно, EJB3.1 (JEE6) поддерживают 2 стиля описания обмена сообщениями для WS:

- request-response
- one-way

Если в <operation> присутствует <input>, за ним <output> и, необязательно, множество <fault>, то это request-response операция. В примере выше операция makeReservation является именно такой.

Если же в <operation> есть только <input>, то это one-way операция.

```
Ex: пример one-way операции WS

<operation name="submitReservation">
  <input message="titan:ReservationMessage"/>
</operation>
```

Обмен сообщениями request-response соответствует стилю RPC: ответ на запрос. Обмен one-way соответствует асинхронному сообщению, когда клиент отправляет сообщения, но не ждет ответа на него. Также one-way обмен используется для доставки XML-документов, поскольку там не требуется ни параметров, ни результата.

Оба типа обмена сообщениями могут использовать в формате Document/Literal и RPC/Literal.

Кроме того существуют еще 2 дополнительных типа обмена сообщениями, поддерживаемые спецификацией WSDL, но не WS-I Basic Profile 1.1 и JAX-RPC:

- notification (единственный <output>)
- solicitation (<output> после которого идет <input>)

Обмен типа notification аналогичен прослушиванию событий. Обмен типа solicitation аналогичен callback-вызову.

Элементы <message> описывают параметры и выходной результат операций. Они связаны через свой атрибут name с соответствующими операциями при помощи целевого пространства имен WSDL (префикс titan в примере):

```
<message name="RequestMessage">...</message>
  <operation name="makeReservation">
    <input message="titan:RequestMessage"/>
    ...
  </operation>
</portType>
```

Элемент <types>

Этот элемент является непосредственным потомком <definitions>. И позволяет вводить сложные XML-типы и XML-элементы при помощи вложенных или импортированных схем.

Для формата сообщений RPC/Literal вместо нескольких параметров <part> встроенных XML-типов сообщения <message> из примера выше, можно определить и использовать тип-структуру, которая описывала бы все параметры операции.

Ex: пример определения сложного типа-обертки для формата RPC/Literal

```
<?xml version="1.0"?>
<definitions ...>
  <types>
    <xsd:schema targetNamespace="http://www.titan.com/TravelAgent">
      <xsd:complexType name="ReservationType">
        <xsd:sequence>
          <xsd:element name="cruiseId" type="xsd:int"/>
          <xsd:element name="cabinId" type="xsd:int"/>
          <xsd:element name="customerId" type="xsd:int"/>
          <xsd:element name="price-paid" type="xsd:double"/>
        </xsd:sequence>
      </xsd:complexType>
  </xsd:schema>
  </types>
```

```
…
 <message name="RequestMessage">
   <part name="reservation" type="titan:ReservationType" />
 </message>
…
<portType name="TravelAgent">
 <operation name="makeReservation">
   <input message="titan:RequestMessage"/>
…
 </operation>
</portType>
<binding name="TravelAgentBinding" type="titan:TravelAgent">
 <soap:binding style="rpc" … />
 …
</binding>
</definitions>
```

Тут в пространстве имен WSDL-документа при помощи вложенной XSD создается сложный тип ReservationType, который затем указан в качестве единственного параметра RequestMessage.

Элемент <types> часто используется в обмене XML-документами, поскольку позволяет задавать в качестве отправляемого и принимаемого сложные XML-документы на основе вложенных схем. Для обычно используемого в этих целях стиля сообщений Document/Literal определяются простые или сложные XML-элементы, а не типы. При проектировании обмена сообщениями в стиле Document/Literal используются 2 шаблона описания операций - Bare и Wrapped, которые WSDL-парсер распознает по характерным признакам.

При использовании шаблона Document/Literal Bare, имена xxx параметров каждой операции будут видны в WSDL-документе, конкретнее, они должны быть указаны в атрибутах name соответствующих частей сообщений: <part name="xxx"...>. Результату должен соответствовать <part name="yyyResponse"...>, где yyy – имя операции. Если у операции нет параметров или результата, то все равно создаются соответствующие пустые элементы <message>. WS-I Basic Profile 1.1 требует наличия единственного прямого потомка SOAP-элемента <Body>, следовательно, в шаблоне Bare допускается только один элемент <part> и операции с единственным аргументом. Обойти это ограничение можно лишь запихнув остальные параметры в SOAP-элемент <Header>.

Ex: фрагменты wsdl для Document/Literal/Bare, описывающие однопараметрическую функцию f(x), с векторным аргументом x (класс с целочисленными полями x0 и x1) и строковым результатом:

```
<?xml version="1.0"?>
<definitions
 xmlns:tns = "http://example.com/example-ws"
 targetNamespace = "http://example.com/example-ws"
...>
```

```
 <types>
  <xsd:schema targetNamespace="http://example.com/example-ws">
   <xsd:element name="fReq" type="RequestType"/>
   <xsd:element name="fResp" type="xsd:string"/>
   <xsd:complexType name="RequestType">
    <xsd:sequence>
     <xsd:element name="x0" type="xsd:int"/>
     <xsd:element name="x1" type="xsd:int"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:schema>
 </types>
…
 <message name="RequestMessage">
  <part name="x" element="tns:fReq" />
 </message>
 <message name="ResponseMessage">
  <part name="fResponse" element="tns:fResp" />
 </message>
 <portType name="exWS">
  <operation name="f">
   <input message="tns:RequestMessage"/>
   <output message="tns:ResponseMessage">
  </operation>
 </portType>
 <binding ...>
  <soap:binding style="document" .../>
…
 </binding>
…
</definitions>
```

Шаблон Document/Literal Wrapped является функциональным гибридом RPC/Literal и Document/Literal : с одной стороны, все параметры описываются элементами в единой схеме, что делает SOAP-сообщение валидным XML-документом; с другой стороны, как и в RPC, элементы параметров должны быть обернуты в корневой элемент, одноименный с соответствующей операцией, что допускает многопараметрические операции и хорошо автоматизируется при автогенерации WSDL-документа. Каждый элемент <message> использует только один элемент <part>, в котором указывается элемент-обертка. Этот элемент-обертка содержит элементы параметров соответствующей операции. Имя элемента-обертки входных параметров должно совпадать с именем операции. Имя элемента-обертки выходных параметров задается как “yyyResponse”, где yyy – имя операции. Если параметры или результат пусты, то все равно создается элемент не содержащего элементов типа. Для всех элементов <part> атрибутам name должно быть задано значение "parameters": <part name="parameters"...>.

Ex: фрагменты wsdl для Document/Literal/Wrapped, описывающие однопараметрическую функцию f(x), с векторным аргументом x (класс с целочисленными полями x0 и x1) и строковым результатом:

```
<?xml version="1.0"?>
<definitions
 xmlns:tns = "http://example.com/example-ws"
 targetNamespace = "http://example.com/example-ws"
...>
 <types>
  <xsd:schema targetNamespace="http://example.com/example-ws">
   <xsd:element name="f" type="RequestType"/>
   <xsd:element name="fResponse" type="xsd:string"/>
   <xsd:complexType name="RequestType">
    <xsd:sequence>
     <xsd:element name="x0" type="xsd:int"/>
     <xsd:element name="x1" type="xsd:int"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:schema>
 </types>
…
 <message name="RequestMessage">
  <part name="parameters" element="tns:f" />
 </message>
 <message name="ResponseMessage">
  <part name="parameters" element="tns:fResponse" />
 </message>
 <portType name="exWS">
  <operation name="f">
   <input message="tns:RequestMessage"/>
   <output message="tns:ResponseMessage">
  </operation>
 </portType>
 <binding ...>
  <soap:binding style="document" .../>
…
 </binding>
…
</definitions>
```

Элемент <binding>

Элемент <binding> определяет тип кодировки и протокол, на котором основывается обмен SOAP-сообщениями. Каждый эемент <binding> связан с конкретным типом порта (интерфейса) WS <portType>.

```
<?xml version="1.0"?>
<definitions ...>
...
<portType name="TravelAgent">
…
</portType>
```

```
<binding name="TravelAgentBinding" type="titan:TravelAgent">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="makeReservation">
    <soap:operation soapAction="" />
    <input>
      <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
    </input>
    <output>
      <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>
    </output>
  </operation>
</binding>
...
</definitions>
```

Элементы <binding> всегда связаны с протокол-специфическими элементами — обычно элементами, описывающими привязку протокола SOAP (фактически это единственная привязка, допустимая WS-I Basic Profile 1.1). Поскольку JEE WS должны поддерживать SOAP с вложениями, также поддерживается и MIME-привязка, когда вместе с SOAP-сообщениями посылаются вложенные картинки, документы и т.д. Это отдельная тема, которая здесь не рассматривается.

Подобно <portType>, элемент <binding> содержит элементы <operation>, <input>, <output>, <fault>. Фактически <binding> является подробным описанием конкретного <portType>: его элементы подробно описывают аналогичные элементы <portType>. В предыдущем примере использовался HTTP-протокол с RPC/Literal стилем обмена сообщениями. Привязка WSDL к Document/Literal стилю несколько отличается:

```
<binding name="TravelAgentBinding" type="titan:TravelAgent">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="submitReservation">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>
```

В примере элемент <binding> описывает стиль Document/Literal одностороннего обмена с WS, заданного в операции <operation name="submitReservation"> типа порта <portType name="TravelAgent">.

Элемент <service>

Веб-служба должна иметь точки входа, по которым ее можно вызвать. Они называются конечным точками веб-службы (Endpoint WS). Каждая конечная точка веб-службы описывается URL доступа, протоколом обмена, привилегиями и т.д.

Элемент <service> описывает WS как совокупность конечных точек. Он содержит элементы <port>, каждый из которых описывает конечную точку веб-службы: задает порт (интерфейс веб-службы) конкретного типа, порядок сетевого обмена и URL доступа к порту.

```
<service name="TravelAgentService">
  <port name="TravelAgentPort" binding="titan:TravelAgentBinding">
    <soap:address location="http://www.titan.com/webservices/TravelAgent" />
  </port>
</service>
```

В примере <service> описывает WS TravelAgentService с конечной точкой TravelAgentPort, доступной по URL http://www.titan.com/webservices/TravelAgent. Тип порта этой конечной точки - TravelAgent и порядок сетевого обмена задаются в связанном элементе <binding name="TravelAgentBinding" ...>.

UDDI 2.0

Universal Description Discovery & Integration (универсальное описание, обнаружение и встраивание) — стандарт, описывающий обнаружение и публикацию WS в сети. Это в основном репозитории со статичной структурой данных, описание организаций и WS, которые они представляют.

UDDI не является фундаментальной частью для SOAP, WSDL и XML. Но оно рассматривается как базовый компонент для JEE WS.

Для описания UDDI часто используют аналогию с white/yellow/green электронными страницами. Можно искать организацию по имени или ID (white pages). Можно искать по роду деятельности или продукции (yellow pages). Можно изучать технические элементы UDDI и тем самым получать информацию о WS (green page). Т.е. UDDI – каталог, который позволяет рекламировать свои и искать чужие WS.

UDDI-реестры не только обеспечивают информацию о WS и хостах. UDDI-реестры собственно являются WS. Они позволяют запрашивать, искать, удалять, изменять информацию в UDDI-реестре при помощи набора SOAP-сообщений. Каждый UDDI-совместимый продукт должен поддерживать стандартные структуры данных UDDI и набор SOAP-сообщений для запросов UDDI-реестра.

Помимо приватных UDDI-реестров, которые могут быть заведены организациями существуют Universal Business Registry (UBR), доступные всем. Эти реестры запрашиваются с одного из 4 сайтов, запущенных IBM, SAP, NTT и Microsoft. При размещении информации на одном из этих сайтов будет произведена репликация на остальные.

От стандартов к реализации

Понимание фундаментальных стандартов для WS (XML, SOAP, WSDL) необходимо для становления квалифицированного разработчика WS. Но для этого также необходимо понять, как реализованы WS в ПО.

Множество платформ (JEE, .NET, Perl) позволяют строить рабочие системы, основываясь на стандартах WS. Далее будет рассмотрены WS в JEE, особенно поддержка в EJB. Для поддержки WS в EJB используется JAX-WS.

Описание JAX-WS

Тут будет дано схематичное описание веб-службы JAX-WS и ее клиентов, во многом основанное на наследии JAX-RPC. Прикладное использование JAX-WS, основанное на улучшенной автоматизации через аннотирование, а также детали реализации рассматриваются в другом разделе.

Реализация веб-службы, готовая работать на сервере, называется компонентом веб-службы (компонентом WS). Для создания компонентов SOAP WS в JSE и JEE имеется определенный набор спецификаций.

JAX-WS (Java API for XML WS) – прикладной Java-интерфейс основанных на XML веб-служб, ядро стандарта SOAP WS в JSE и JEE. Он позволяет создавать компоненты SOAP WS и клиентский код к ним.

Основной задачей API веб-служб было соединение Java-объектов со стандартными протоколами WS. В отличие от других распределенных протоколов, WS были спроектированы гибкими и расширяемыми. Это сделало технологию WS более адаптивной, но вместе с тем усложнило достижение незаметности (прозрачности) ее использования для пользователей. Объявление и вызов компонента WS в Java должно быть как можно ближе к созданию и вызову обычных POJO. Но хороший API для WS в результате достижения подобной прозрачности становится весьма объемным. В JAX-WS удалось снизить сложность и улучшить прозрачность по сравнению с предшественниками.

API JAX-WS вместе с эталонной реализацией переехала в JSE, начиная с версии 2.0, перенесенной в JSE6. В JSE8/JEE7 JAX-WS описывает спецификация JAX-WS 2.2A (JSR 224). Изначально поддержка SOAP WS была основана на спецификации JAX-RPC, позднее JAX-RPC переименовали в JAX-WS из-за ошибочного мнения о том, что WS в JEE охватывают лишь технологию удаленного вызова процедур, а не обмен XML-документами. В JEE6 поддержка SOAP WS была основана на спецификациях JAX-WS 2.1 и его предшественника - JAX-RPC 1.1. С JEE7 остался уже только JAX-WS.

Rem: В JEE7 поддержка SOAP WS не интегрирована с управляемыми CDI-компонентами и технологией Bean Validation.

Помимо прочего, к поддержке SOAP WS относится SAAJ и JAXR. Также JAX-WS зависит от других технологий, например, от JAXB 2.2 (JSR 222).

- JAX-WS похож на CORBA и RMI за исключением использования протокола SOAP.
- SAAJ — API управления структурой SOAP-сообщений.
- JAXR — API получения доступа к реестрам SOAP WS, обычно UDDI.
- JAXB – фреймворк, осуществляющий связывание XML-разметки и Java-объектов.

JAX-WS позволяет экспортировать в виде компонентов веб-служб POJO, снабженные метаинформацией (аннотациями). Для развертывания веб-служб нужен сервер, поддерживающий JAX-WS. Это может быть любой Web-сервер, поддерживающий JAX-WS, в т.ч. встроенный в JSE с 6-й версии HTTP-сервер. Также подходят EJB-контейнер или Web-контейнер любого JEE-сервера приложений.

В Web-модели, т.е. при развертке на обычном Web-сервере (в т.ч. в JSE) или в Web-контейнере JEE-сервера, для реализации компонентов JAX-WS используется POJO, называемый сервлетом конечной точки (Servlet Endpoint). В EJB-модели для непосредственной реализации WS используется EJB без сохранения состояния (SLSB и @Singleton), называемый EJB конечной точки (EJB Endpoint).

Rem: EJB Lite контейнер JEE-сервера не поддерживает JAX-WS, поэтому компонент веб-службы может быть развернут лишь в Web-модели как Servlet Endpoint.

JAX-WS помимо серверной части предоставляет и API для создания клиентского кода, в котором также можно использовать аннотации.

В JSE начиная с 6 версии присутствует эталонная реализация JAX-WS RI, использующая встроенный HTTP-сервер (com.sun.net.httpserver). Известными вендорными JEE реализациями стандартов WS являются Metro (включена в Glassfish), Apache CXF и Apache Axis 2.

Компоненты WS и клиенты при развертке получают различные возможности в зависимости от модели. Самыми простыми будут веб-службы на основе JSE (JAX-WS RI), веб-службы на основе EJB будут обладать наибольшими возможностями:

Свойства веб службы JAX-WS	JSE Web модель	JEE Web модель	JEE EJB модель
Является POJO	+	+	+
Внедрение зависимостей	+	+	+
Поддержка событий жизненного цикла	-	+	+
Декларативное управление транзакциями	-	-	+

Декларативное управление безопасностью	-	-	+
Использование перехватчиков	-	-	+
Использование службы таймеров	-	-	+

Основные пакеты JAX-WS (API JSE):

- javax.xml.ws и javax.xml.ws.* – основное API JAX-WS
- javax.jws и javax.jws.* - описание аннотаций, связывающих WSDL и Java-реализацию. Для JSE8 это спецификация WS-Metadata 2.3 (JSR 181).

Ниже будет дано введение в JAX-WS. Полное описание WS заняло бы 500-страничную книжку.

Пути описания JAX-WS веб-служб

Артефакты, описывающие веб-службу JAX-WS можно поделить на 2 группы: XML-документы (WSDL-документ и связанные с ним схемы), Java-реализацию (Java-класс функциональной реализации самой WS, связанные с ним интерфейсы, вспомогательные классы). И XML и Java-реализация способны полностью описать веб-службу JAX-WS.

WSDL-описание (контракт) может использоваться для создания Java-реализации как потребителя, так и поставщика. Такой подход называется нисходящим. В JDK существует инструмент wsimport для автоматической генерации по WSDL-документу Java-кода, полностью реализующего обмен сообщениями как на клиентской, так и на серверной стороне: SEI, SEIC (заготовка), классы исключений, классы JAXB-типов, асинхронные компоненты объекта. После применения wsimport, для завершения создания Java-реализации компонента WS останется лишь дописать его логическое содержание в созданных заготовках и связать со сгенерированными интерфейсами. При использовании сложных входных и выходных сообщений, полученный код может быть неоптимальным.

Rem: для генерации Java-кода по отдельным XSD можно также использовать непосредственно JAXB-компилятор xjc, входящий в состав JDK.

Другой подход описывает уже существующую реализацию WS в терминах контракта (WSDL). Такой подход называют восходящим. В JDK существует инструмент wsgen для создания по Java-реализации (JAX-WS Endpoint) XML-документации. Процессом такого документирования можно управлять при помощи метаинформации. Если в качестве параметров или выходного значения методов реализации использовались сложные типы, то полученные XML-схемы могут быть несовместимы с некоторыми языками. Могут быть проблемы с Java Generics (?).

Rem: при необходимости передавать методу в качестве параметра или получать от него обратно объект сложного типа, можно создавать упрощенную копию этого объекта (DTO),

избавленную от каскадов наследования, коллекций и т.п.

Создание WSDL-документа и Java-реализации вручную наиболее сложный вариант, зато более гибкий и потенциально эффективный.

Схема запросов через JAX-WS

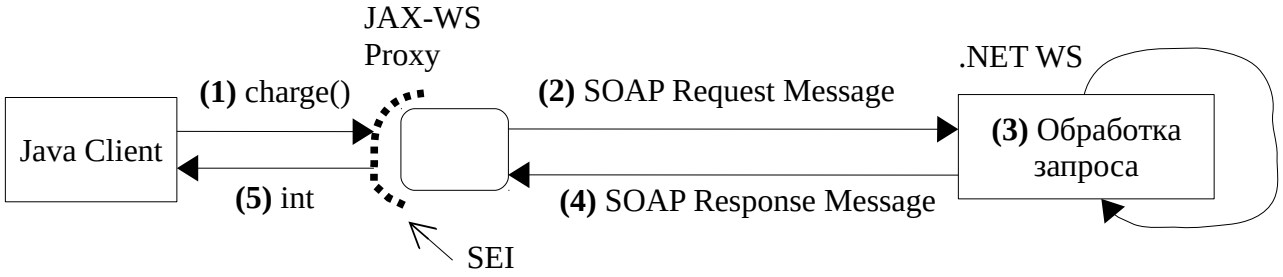
Основным понятием при запросе веб-службы является ее конечная точка (WS Endpoint), в общем случае это один из URL, по которому можно обратиться к WS, поскольку каждый WS может иметь множество точек входа, различающихся привилегиями, протоколами.

В JAX-WS имеются различные Java-представления конечной точки веб-сервиса (JAX-WS Endpoint). Класс реализации конечной точки веб-службы (Service Endpoint Implementation Class, SEIC) – Java-класс, в котором реализуются операции веб-службы. Интерфейсное представление дает SEI (Service Endpoint Interface - интерфейс конечной точки), являющийся Java-интерфейсом, в котором перечислены методы, реализующие операции соответствующего порта веб-службы (WSDL-элемент <portType>).

JAX-WS обеспечивает программную модель клиента, которая позволяет запрашивать WS на других платформах. Иными словами Java-клиенты при помощи JAX-WS могут запрашивать по сети WS, базирующиеся как на Java-платформе, так и на других (.NET, Perl, C++). Существуют 2 типа клиентского API для запросов JAX-WS:

- Proxy Client API (API клиентского посредника)
- Dispatch Client API (API клиентского диспетчера)

Модель клиентского посредника (proxy client) похожа на программную модель CORBA или Java RMI. В ней клиент запрашивает удаленную службу (конечную точку WS) посредством обращения к удаленному интерфейсу Service Endpoint Interface (SEI, интерфейс конечной точки), реализованному в прокси (клиентской сетевой заглушке в терминологии CORBA). Эта заглушка незаметно для клиента, который обращается к ней, как к обычному объекту, переводит запросы к SEI в сетевые сообщения, которые посылаются удаленной службе. При этом все артефакты для создания прокси на стороне клиента должны быть созданы заранее при помощи WSDL-документа. Это похоже на использование удаленных ссылок на EJB, за исключением того, что используется SOAP поверх HTTP вместо CORBA IIOP. А также того, что в JAX-WS не требуется отслеживать идентичность UID (serialVersionUID) классов объектов, которыми обмениваются клиент и сервер, поскольку вместо низкоуровневой сериализации передаваемых объектов тут используется их маршалинг в XML-структуру.



Цикл выполнения запрос-ответ в модели клиентского прокси JAX-WS почти такой же, как в RMI. На шаге (1) клиент вызывает JAX-WS прокси, который реализует интерфейс конечной точки (SEI) WS. Вызов метода трансформируется JAX-WS прокси в SOAP-сообщение, которое и посылается удаленной службе на шаге (2). На шаге (3) WS обрабатывает SOAP-запрос и посылает на шаге (4) SOAP-ответ. На шаге (5) JAX-WS прокси трансформирует полученный SOAP-ответ либо в возвращаемое значение, либо в исключение (если ответ содержит SOAP fault) и передает клиенту.

Модель клиентского диспетчера (dispatch client) создана для общения с WS на уровне отдельных XML-сообщений. При этом никаких заранее созданных артефактов не требуется, требуется знание структуры самих сообщений и связанных с ними объектных типов Java. В отличие от модели прокси все артефакты для связи с WS создаются клиентом динамически (в run-time) при помощи вызова статических методов Dispatch Client API, вместо объекта прокси создается объект диспетчера, который и выполняет низкоуровневую обработку и создание XML-сообщений. Разбор XML-сообщений в данной модели может проводиться клиентом без схемы и JAXB при помощи JAXP. В целом технически схема обмена аналогична модели прокси.

Rem: в JAX-RPC существовало 3 клиентских API: генерируемые заглушки (Generated Stubs); динамические прокси (Dynamic Proxies); динамический интерфейс вызова (Dynamic Invocation Interface — DII). В JAX-WS перекочевали динамические прокси и замена DII – клиентский диспетчер.

Обработка вызова JAX-WS веб-сервиса на стороне сервера

JAX-WS использует HTTP-транспорт, поэтому для реализации обмена сообщениями используется сервлетная инфраструктура. Т.е. в качестве прокси на стороне сервера используется некий сервлет, слушающий URL конечной точки и отвечающий за ответ. Это, помимо прочего, позволяет использовать готовые возможности Web-контейнера, например, базовую аутентификацию и авторизацию без обращения к WS-Security. Последовательность действий на стороне сервера имеет следующий вид:

- получение SOAP-запроса системным сервлетом, играющим роль прокси конечной точки и слушающим ее URL
- создание объекта MessageContext - контекста сообщения SAAJ, из которого можно получить информацию о текущем запросе, получить доступ к заголовку и телу SOAP и т.д.
- вызов цепи обработчиков входного сообщениями
- определение целевой WSDL-операции и соответствующего Java-метода SEIC
- unmarshaling (разупорядочивание) входных параметров в Java-объекты JAXB-

компилятором
- вызов Java-метода реализации и возврат Java-ответа в виде объекта или исключения
- обновление объекта MessageContext
- вызов цепи обработчиков исходящего сообщениями
- marshaling ответа в SOAP-сообщение и возврат его по транспортному протоколу

Вызов JAX-WS веб-сервисов JSE-клиентом

Последовательность действий на стороне клиента похожа на серверную:

- создание экземпляра прокси, реализующего SEI
- вызов конечной точки JAX-WS
- маршалинг Java-параметров и создание SOAP-запроса вызываемой операции
- вызов цепи обработчиков сообщениям
- отправка сообщения по транспортному протоколу
- получение ответного SOAP-сообщениями
- unmarshaling SOAP-ответа в Java-объекты

Основной интерфейс, который описывает веб-службу JAX-WS, называется интерфейсом конечной точки (SEI). JAX-WS совместимый компилятор генерирует для клиента интерфейс конечной точки, используя элемент <portType> WSDL. Этот интерфейс, будучи скомбинированным с данными WSDL-элементов <binding> и <port>, используется для создания клиентских динамических прокси в deploy-time. Организация, которая разворачивает WS, предоставляет WSDL-документ с его описанием.

Рассмотрим пример. Пусть есть компания Titan Cruises, которая заключила субдоговор с компанией Charge-It, Inc., для обслуживания платежей, сделанных клиентами, использующими кредитные карты. Charge-It запускает систему на .NET и предоставляет клиентам приложение, обрабатывающее кредитные карты, в виде WS. WSDL-документ описывает WS.

Ex : WSDL для веб-службы Charge-It

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://charge-it.com/Processor"
   targetNamespace="http://charge-it.com/Processor">
 <message name="chargeRequest">
   <part name="name" type="xsd:string"/>
   <part name="number" type="xsd:string"/>
   <part name="expDate" type="xsd:dateTime"/>
   <part name="cardType" type="xsd:string"/>
```

```
   <part name="amount" type="xsd:float"/>
  </message>
  <message name="chargeResponse">
    <part name="return" type="xsd:int"/>
  </message>
  <portType name="Processor">
   <operation name="charge">
     <input message="tns:chargeRequest"/>
    <output message="tns:chargeResponse"/>
  </operation>
  </portType>
  <binding name="ProcessorSoapBinding" type="tns:Processor">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="charge">
      <soap:operation soapAction="" style="rpc"/>
     <input>
       <soap:body use="literal" namespace="http://charge-it.com/Processor"/>
     </input>
     <output>
       <soap:body use="literal" namespace="http://charge-it.com/Processor"/>
     </output>
   </operation>
  </binding>
  <service name="ProcessorService">
   <port name="ProcessorPort" binding="tns:ProcessorSoapBinding">
     <soap:address location="http://www.charge-it.com/ProcessorService"/>
   </port>
  </service>
</definitions>
```

Интерфейс конечной точки WS Charge-It связан с WSDL-элементом <portType> и соответствующими ему элементами <message>. По-умолчанию по этим данным будет сгенерирован примерно следующий интерфейс (аннотации для простоты опущены):

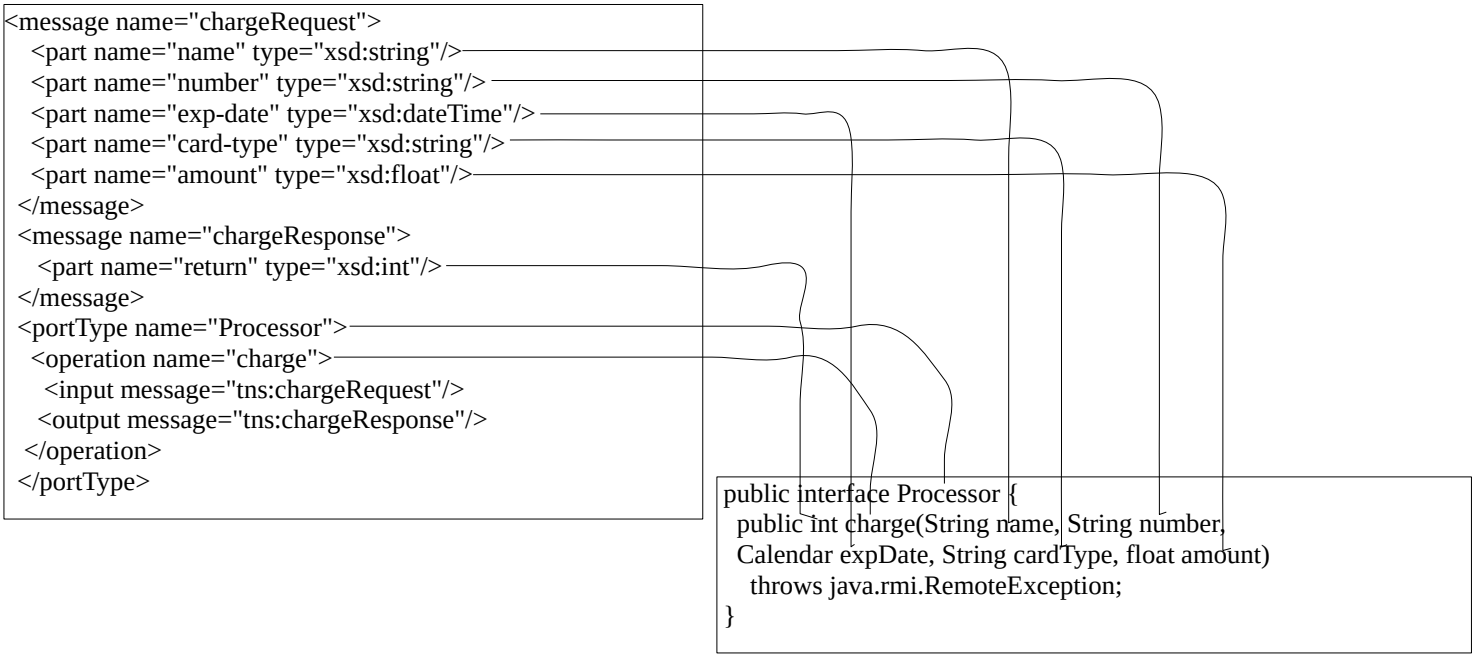
Ex : интерфейс конечной точки (SEI) Charge-It, соответствующий WSDL

```
package com.charge_it;
public interface Processor {
public int charge(String name, String number, java.util.Calendar expDate, String cardType, float amount)
;
}
```

Имена методов, имена и типы параметров, исключений выводятся из WSDL-документа в соответствующую метаинформацию (аннотации или дескриптор) SEI. Любой метод может пробрасывать исключения, которые будут затем обрабатываться движком JAX-WS и передаваться клиенту в SOAP-элементе <Fault>.

Реализация SEI не требуется, это дело инструментов (вроде wsimport), создающих прокси-заглушки. Соответствие между <portType>, <message> и интерфейсом конечной

точки WS задается метаданными, однако по-умолчанию задаются и соответствующие Java-имена. Ниже показано соответствие Java-имен по-умолчанию в SEI и WSDL (метаинформация для простоты опущена):



Название интерфейса конечной точки «Processor» берется из атрибута name элемента <portType>. Названия методов интерфейса конечной точки выводятся из элементов <operation>, в примере единственный метод - «charge». Параметры методов интерфейса конечной точки выводятся из атрибута message элемента <input>, связанного посредством пространства имен WSDL с элементом типа сообщения <message>, который и содержит в подэлементах <part> типы и имена параметров. Аналогично параметрам выводится и тип возвращаемого значения — через атрибут message элемента <output>, в примере это int.

Спецификация JAX-WS определяет точное соответствие между некоторыми встроенными типами XML-схемы, примитивами и классами Java. Т.е. то, как типы XML-схемы, объявленные в элементах <part> WSDL будут отображены на параметры и возвращаемое значение методов интерфейса конечной точки.

Встроенные типы XML (XS-типы)	**Типы Java**
xsd:base64Binary byte []	byte []
xsd:QName	java.xml.namespace.QName
xsd:decimal	java.math.BigDecimal
xsd:integer	java.math.BigInteger
xsd:dateTime	java.util.Calendar
xsd:string	java.lang.String
xsd:double	Double
xsd:float	Float

xsd:long	Long
xsd:int	Integer
xsd:short	Short
xsd:Boolean	Boolean
xsd:byte	Byte
xsd:hexBinary	byte []

JAX-WS также определяет отображение nillable-типов XS в оболочки примитивов: xsd:int — Integer, xsd:double — Double и т.д.

В дополнение ко всему этому, JAX-WS определяет связь сложных типов, определенных во вложенных схемах WSDL-элемента <types> с классами JAXB.

Клиентский динамический прокси (заглушка), реализующий интерфейс конечной точки, генерируется из описания элементов <binding> и <part>. JAX-WS переводит определение стиля обмена сообщениями из <binding> в алгоритм упорядочивания (marshaling), который конвертирует все вызовы методов этой заглушки конечной точки в запросы и ответы SOAP.

Ex : фрагмент WSDL-документа с <binding>

```
<binding name="ProcessorSoapBinding" type="tns:Processor">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="charge">
    <soap:operation soapAction="" style="rpc"/>
    <input>
      <soap:body use="literal" namespace="http://charge-it.com/Processor"/>
    </input>
    <output>
      <soap:body use="literal" namespace="http://charge-it.com/Processor"/>
    </output>
  </operation>
</binding>
```

В данном примере, в соответствии с описанием из <binding>, WS использует сообщения RPC/Literal SOAP 1.1 в режиме request-response. Динамический прокси ответственен за конвертацию вызовов к интерфейсу конечной точки в SOAP-сообщения, посылаемые затем WS. Он также ответственен за обратную конвертацию SOAP-сообщений, присылаемых в ответ WS, в возвращаемое значение, либо в исключение, если SOAP-сообщение содержит элемент <fault> .

Динамический прокси также анализирует WSDL-элемент <port>, где определен сетевой адрес, по которому расположен WS.

Ex : фрагмент WSDL-документа с <port>

```
<service name="ProcessorService">
  <port name="ProcessorPort" binding="tns:ProcessorSoapBinding">
    <soap:address location="http://www.charge-it.com/ProcessorService"/>
  </port>
</service>
```

В элементе <soap:address> в атрибуте location задан адрес с которым прокси будет обмениваться SOAP-сообщениями.

В дополнение к интерфейсу конечной точки WS, компилятор JAX-WS создает для клиента и класс службы (Service-класс) для получения экземпляра прокси в run-time. Этот класс расширяет javax.xml.ws.Service и генерируется из WSDL-элемента <service> (см. в предыдущем примере). В результате получится примерно следующее (имена по-умолчанию, метаинформация опущена):

Ex : пример фрагмента класса службы

```
package com.charge_it;
  public class ProcessorService extends javax.xml.ws.Service {
    …
    public com.charge_it.Processor getProcessor( ) {
      return super.getPort(…);
    }
    ...
}
```

В этом примере сгенерированный Service-класс ProcessorService предоставляет методы getProcessor() для получения прокси интерфейса Processor, готового запрашивать WS.

Вызов JAX-WS EJB-клиентом

Вызов из EJB по сути мало отличается от вызова JSE-клиентом, есть лишь некоторые особенности.

Также как и другие управляемые ресурсы (JDBC, JMS и т.д.), клиентские артефакты JAX-WS веб-служб могут быть внедрены в свойства EJB. Также они могут быть связаны с JNDI ENC – именами в deploy-time. Спецификация JAX-WS определяет набор аннотаций для внедрения SEI (заглушек) и классов служб (javax.xml.ws.Service).

Rem: в предшествующем JAX-RPC для внедрения артефактов использовался дескриптор развертывания (ejb-jar.xml), где был определен элемент <service-ref>. В JAX-WS используются только аннотации, хотя в некоторых реализациях для внедрения можно также использовать вендор-специфические XML-дескрипторы. Например, Glassfish 4 определяет собственную XML-структуру <service-ref> в дескрипторах glassfish-.xml.*

Интерфейсы конечных точек веб-служб (SEI) также могут инициализироваться заглушками для общения с удаленным WS через объекты класса службы: после получения объекта Service можно получить заглушку для нужного порта (интерфейса WS) соответствующим его get-методом.

Ex : пример использования объекта Service и интерфейса конечной точки (SEI) для WS

```
package com.titan.travelagent;
import com.charge_it.Processor;
import com.charge_it.ProcessorService;
...
@Stateful
public class TravelAgentBean implements TravelAgentRemote {
…
  @WebServiceRef
  private ProcessorService processorService;
  ...
  public TicketDO bookPassage(CreditCardDO card, double price) … {
    ...
    java.util.Calendar expDate = new Calendar(card.date);
    Processor processor = processorService.getProcessor( );
    processor.charge(customerName, card.number, expDate, card.type, price);
    ...
 }
  ...
}
```

В примере объект службы ProcessorService внедрен в SFSB в поле processorService. При помощи метода getProcessor( ) создается заглушка интерфейса конечной точки (SEI), в примере это интерфейс Processor.

Вызов метода SEI через прокси внутри транзакционного метода EJB может вызвать проблемы. Дело в том, что вызовы WS по-умолчанию не относятся к транзакционному контексту вызывающей стороны. Если прокси сталкивается с сетевым сбоем или проблемой обработки SOAP-сообщения, то он выбрасывает RemoteException, которое затем оборачивается в EJBException, вызывая откат целой транзакции. Но если сбой произошел после того, как WS отработал, но до того, как завершился EJB-метод, то произойдет частичный откат транзакции: изменения, сделанные непосредственно в EJB будут отменены, но все то, что сделал WS - останется.

Создание WS с помощью JAX-WS

В JSE8 (эталонная JAX-WS RI) и JEE7 обеспечивается различные программные модели для создания компонентов JAX-WS: Web-модель и EJB-модель. Различаются они только реализацией, а описываются одинаково при помощи метаинформации. Для описания применяется стандартное конфигурирование исключениями, при котором задается поведение по-умолчанию, которое в случае необходимости можно корректировать.

Ключевой компонент в JAX-WS – класс реализации конечной точки веб-службы (SEIC), где собственно и содержатся методы, представляющие операции веб-службы. Это POJO-класс (в т.ч. EJB), удовлетворяющий следующим требованиям:

- класс аннотирован @javax.jws.WebService или @javax.jws.WebServiceProvider или описан в дескрипторе webservices.xml
- класс должен подходить для встраивания в фреймворк, т.е. определен как public, не иметь спецификаторов final и abstract и обладать публичным общедоступным конструктором по-умолчанию (без аргументов) и не определять метод finalize()
- объект не должен каким-либо образом сохранять состояние клиента

В дополнение к SEIC используется уже упоминавшийся ранее SEI. Для реализации JAX-WS компонента этот интерфейс уже необязателен, поскольку автоматически выводится из SEIC, но иногда полезен. SEI определяет абстрактное соглашение (контракт) об использовании веб-службы, которое реализуется в SEIC. При этом SEI связан с SEIC метаинформацией, прямая реализация этого интерфейса SEIC необязательна.

Согласно спецификации JSR 181 (WS-Metadata), класс, удовлетворяющий перечисленным требованиям, является SEIC для Web-модели и называется также сервлетом конечной точки (Endpoint Servlet).

Если же такой класс конечной точки помечен дополнительно как SLSB или @Singleton, то он является EJB конечной точки (EJB Endpoint), сочетающим свойства WS и сессионного бина без поддержки состояния в EJB-модели. Объект такого компонента можно вызывать через RMI и SOAP одновременно.

Rem: другие EJB-компоненты не подходят на эту роль. SFSB должен сохранять состояние в сессии, что противоречит архитектуре SOAP WS (запрос-ответ). MDB имеет асинхронную природу, что в общем случае также противоречит архитектуре запрос-ответ. @Singleton допускается, но плохо подходит для обработки параллельных запросов.

EJB Endpoint запрашивается клиентом при помощи SOAP, но тем не менее является полноценным сессионным бином и обладает всеми возможностями EJB:

- запускается в том же EJB-контейнере
- поддерживается теми же окружениями безопасности и транзакций
- обеспечивается доступом к другим EJB и ресурсам через JNDI ENC.

WSDL-документ для создания JAX-WS WS

Каждый SEIC должен быть снабжен WSDL-документом, который описывает WS. Этот WSDL-документ идентичен тому, что используется клиентом веб-службы (точнее полностью охватывает любой из клиентских WSDL-документов). Его можно создать вручную или использовать инструменты, предоставляемые JSE или вендором JEE.

Элемент WSDL <portType> должен быть согласован с SEIC (и SEI, если последний определен). Иными словами связь между WSDL-элементом <portType> и SEIC должна следовать спецификации JAX-WS. Единственный способ 100% гарантировать такую согласованность — сначала сгенерировать корректный WSDL и затем использовать его для создания SEIC WS.

Service Endpoint Interface (SEI)

Интерфейс конечной точки WS – интерфейс, аннотированный @WebService. Процесс создания необязательного SEI для компонента веб-службы по WSDL-документу аналогичен созданию SEI для клиента. Компилятор JAX-WS создает SEI при помощи WSDL-элементов <portType>, <message>, <part>, <types>.

В качестве альтернативы можно начать с написания SEI вручную. А затем получить из него WSDL, используя таблицу соответствия типов Java и встроенных типов XML.

Service Endpoint Implementation Class (SEIC)

Класс реализации конечной точки WS – класс, аннотированный либо @WebService, либо @WebServiceProvider и подходящий для встраивания в фреймворк (расширяемый, не мешающий управлению жизненным циклом): не абстрактный, не финальный, без finalize-метода и с публичным конструктором по-умолчанию.

Для любой JEE-реализации SEIC можно использовать аннотации @PostConstruct и @PreDestroy, управляющие жизненным циклом.

Реализация конечной точки не обязана реализовывать SEI, они связаны при помощи метаинформации. Хотя никто не мешает SEIC напрямую реализовать этот интерфейс.

Ex : пример SEIC - Endpoint SLSB

```
package com.titan.webservice;
import com.titan.domain.*;
import com.titan.cabin.*;
import com.titan.processpayment.*;
import javax.ejb.EJBException;
import java.util.Date;
import java.util.Calendar;
import javax.persistence.*;
@Stateless
@WebService
public class TravelAgentBean implements TravelAgent {
  @PersistenceContext EntityManager em;
  @EJB ProcessPaymentLocal process;
@WebMethod
  public String makeReservation(int cruiseId, int cabinId, int customerId, double price) {
```

```
   try {
     ...
   } catch(Exception e) {
       throw new EJBException(e);
   }
  }
  public CreditCardDO getCreditCard(Customer cust) throws Exception{
    ...
  }
}
```

Java-код SLSB в данном примере ничем не отличается от обычного EJB, он реализует для удобства SEI, хотя это не обязательно. Метаинформация @WebService и @WebMethod дает этому SLSB функциональное отличие, заключающееся в том, что он также отвечает и на вызовы WS вместе с удаленными или локальными вызовами.

Rem: наследовать в методах реализации проброс исключений интерфейса, если таковые есть, по правилам Java необязательно, т.к. ослабление ограничений интерфейса всегда допустимо.

Дескриптор развертки

Для связывания SEIC, SEI и WSDL используется метаинформация. В JAX-WS для этого определена аннотация @WebService. Также спецификация JSR-109 v1.3 допускает совместное использование с ней необязательного уже XML-дескриптора webservices.xml, определяемого в каталоге META-INF.

Дескриптор webservices.xml должен быть упакован вместе с SEIC. Для EJB-модели обычно дескриптор webservices.xml помещается в каталог META-INF jar-архива, но его можно разместить в любом другом месте того же jar-архива, что содержит EJB Endpoint. Для Web-модели WSDL хранится в war-архиве. Для JSE неприменим(?).

Rem: в JAX-RPC использовались также файлы связывания навроде jaxrpc-mapping.xml

Файл webservices.xml может переопределять аннотации @WebService и связывает SEI, SEIC и WSDL-документ.

Ex : пример webservices.xml для JAX-WS в EJB-модели

```
<?xml version='1.0' encoding='UTF-8' ?>
<webservices
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:titan="http://www.titan.com/TravelAgent"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
            http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
  version="1.1">
```

```
    <webservice-description>
      <webservice-description-name>TravelAgentService</webservice-description-name>
      <wsdl-file>META-INF/travelagent.wsdl</wsdl-file>
      <port-component>
        <port-component-name>TravelAgent</port-component-name>
        <wsdl-port>titan:TravelAgentPort</wsdl-port>
        <service-endpoint-interface>
          com.titan.webservice.TravelAgent
        </service-endpoint-interface>
        <service-impl-bean>
          <ejb-link>TravelAgentBean</ejb-link>
        </service-impl-bean>
      </port-component>
    </webservice-description>
</webservices>
```

Каждый элемент <webservice-description> описывает отдельный JAX-WS Endpoint, таких элементов может быть множество.

Элемент <webservice-description-name> - уникальное имя описания веб-службы. Это имя может быть произвольным.

Элемент <wsdl-file> указывает на WSDL-документ, связанный с описываемым JAX-WS Endpoint. Каждый JAX-WS Endpoint может иметь только единственный WSDL-документ. Файл WSDL-документа обычно располагается в каталоге META-INF соответствующего jar или war архива.

Элемент <port-component> привязывает SEIC, описанный либо в ejb-jar.xml, либо в web-xml к WSDL-элементу <port>.

Элемент <port-component-name> описывает логическое имя JAX-WS Endpoint. Оно может быть произвольным.

Элемент <wsdl-port> связывает информацию о развертке JAX-WS Endpoint с WSDL-элементом <port>.

Элемент <service-endpoint-interface> определяет F.Q.N. SEI, это имя должно совпадать с именем того же интерфейса, определенном в элементе <service-endpoint> ejb-jar.xml.

Элемент <service-impl-bean> вместе с дочерним <ejb-link> или <servlet-link> привязывает <port-component> к описываемому SEIC. Значение <ejb-link> должно соответствовать имени EJB в ejb-jar.xml (в примере это TravelAgentBean). Значение <servlet-link> должно соответствовать имени SEIC, прописанному как <servlet-class> в web.xml (хотя SEIC вообще говоря сервлетом не является(?)).

Использование JAX-WS

JAX-WS принес в JSE набор аннотаций, основанный на спецификации JSR-181 (метаданные WS для Java-платформы). Этим метаданные сделали намного более простым процесс объявления WS. Кроме того все аннотации снабжены разумными значениями по-умолчанию, что позволяет проводить конфигурирование методом исключения. Использование метаданных позволяет ослабить связь между контрактом WS (WSDL-документом) и ее реализацией. Выбор аннотаций усиливает типобезопасность по сравнению с текстовыми ID XML-дескрипторов.

В нижеприведенном примере удается обойтись всего 2 аннотациями для превращения SLSB в WS:

```
Ex : пример определения WS при помощи аннотаций JAX-WS

package com.titan.webservice;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
@Stateless
@WebService
public class TravelAgentBean {
   @WebMethod
   public String makeReservation(int cruiseId, int cabinId, int customerId, double price) {
     ...
   }
}
```

Основные аннотации JAX-WS содержатся в 2 пакетах JSE:

- javax.jwx (@WebService, @WebMethod, @WebParam, @WebResult, @OneWay) – аннотации связи WSDL с Java-реализацией. Они настраивают отображение реализации в WSDL и устанавливают связь сигнатуры методов с XML-сообщениями.
- javax.jws.soap (@SoapBinding, @SoapMessageHandler) – аннотации связи протокола SOAP с Java-реализацией. Они настраивают маршалинг объектов в SOAP-сообщения и демаршалинг SOAP-сообщений в объекты.

Представления конечных точек JAX-WS (Endpoint JAX-WS)

Конечная точка — базовое понятие для веб-службы, это адрес (URL), по которому клиент может обратиться к веб-службе.

Основой компонента JAX-WS является реализация его конечных точек (JAX-WS Endpoint). Класс реализации конечной точки веб-службы (Service Endpoint Implementation Class, SEIC) – аннотированный @WebService или @WebServiceProvider Java-класс, в котором реализуются операции веб-службы. Тип SEIC зависит от модели. В EJB-модели это EJB Endpoint (конечная точка EJB) – аннотированный @WebService или @WebServiceProvider EJB-компонент без состояния (@Singleton или SLSB). В Web-

модели (включая JAX-WS RI из JSE) это Servlet Endpoint (конечная точка сервлета) — аннотированный @WebService или @WebServiceProvider POJO.

Интерфейсное представление конечной точки дает SEI (Service Endpoint Interface), являющийся аннотированным @WebService Java-интерфейсом, в котором перечислены методы, реализующие операции соответствующего порта веб-службы (WSDL-элемент <portType>). Клиент использует заглушку, реализующую SEI, как будто это локальный объект. В компоненте JAX-WS явное определение SEI необязательно, так как он автоматически выводится из SEIC или WSDL.

Аннотация @WebService

@javax.jws.WebService – основная аннотация JAX-WS, объявляющая Java-тип как конечную точку (JAX-WS Endpoint). Она применяется к SEIC, SEI и имеет следующие атрибуты:

- name – алиас xxx для JAX-WS Endpoint, соответствующий типу порта <portType name=”xxx”> WSDL-документа. По-умолчанию это короткое имя класса или интерфейса, к которому применяется эта аннотация.
- targetNamespace – пространство имен XML и WSDL, используемое для определения элементов WSDL и XML, которые генерируются по @WebService. По-умолчанию это имя пакета того класса/интерфейса, в котором определена аннотация.
- serviceName – алиас xxx описания конечных точек WS, используемый в <service name=”xxx”> WSDL-документа. Для SEI этот атрибут не применяется, т.к. связан с серверной реализацией конечной точки.
- wsdlLocation – URL WSDL-документа, связанного с WS. Требуется в том случае, когда уже существует WSDL-документ.
- endpointInterface - атрибут используемый для экспорта соглашения (контракта) об использовании WS путем объявления его в виде отдельного Java-интерфейса. Короче это F.Q.N. отделенного SEI.
- portName – алиас xxx описания конкретного порта заданной в serviceName конечной точки WS, соответствующий <service><port name=”xxx”> WSDL-документа. Для SEI не применяется.

Аннотация @WebMethod

@javax.jws.WebMethod применяется для открытия доступа к методам SEI или SEIC. Если в классе нет ни одного @WebMethod метода, то они все включаются в состав порта веб-службы. Это полезно, поскольку WS обычно менее гибки по сравнению со стандартом EJB. Кроме того, хорошей практикой считается ослаблять внешнюю связь между классами, давая доступ только к их минимально-необходимой функциональности.

@WebMethod методы должны быть:

- публичными
- не статическими
- не final
- принимать и выдавать JAXB-совместимые значения

@WebMethod содержит атрибуты для настроек генерируемого WSDL-документа:

- operationName – задает имя xxx WSDL-операции, которую данный метод и реализует. Соответствует <portType><operation name=”xxx”> WSDL-документа. По-умолчанию это символьное имя Java-метода.
- action – задает URI xxx, соответствующий значению <binding><operation><soap:operation soapAction=”xxx”> WSDL-документа. Этот атрибут позволяет компоненту WS находить URI нужного метода просмотром HTTP-заголовка SOAPAction: xxx при POST-запросах вместо поиска имени операции в теле SOAP-сообщения. Потенциально это может привести к небольшому увеличению производительности, в зависимости от используемой реализации JEE.
- exclude – атрибут, используемый для исключения метода из порта WS

Ниже демонстрируется назначение имени операции для Java-метода:

```
Ex : пример определения WS при помощи аннотаций JAX-WS

package com.titan.webservice;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
@Stateless
@WebService(name = "TravelAgent")
public class TravelAgentBean
{
  @WebMethod(operationName = "Reserve")
  public String makeReservation(int cruiseId, int cabinId, int customerId, double price) {
  ...
  }
}
```

Аннотация SOAPBinding

Аннотация @javax.jws.soap.SoapBinding позволяет настраивать стиль WS, который используется Endpoint WS. Аннотация применяется к типу (SEIC/SEI) и содержит следующие атрибуты:

- style = {DOCUMENT, RPC}
- use = {LITERAL, ENCODED}
- parameterStyle = {BARE, WRAPPED}

Важно отметить, что в JEE7 всегда use = LITERAL, т.к. ENCODED запрещен в WS-I Basic Profile 1.1 и не поддерживается JAX-WS . По-умолчанию и/или при отсутствии аннотации @SoapBinding используется стиль DOCUMENT/LITERAL/WRAPPED.

Ниже таблица допустимых комбинаций атрибутов стилей:

style	use	parameterStyle	описание
RPC	LITERAL	N/A	Каждый параметр операции WS связывается с wsdl:part, который связан с определением XML-схемы типа
DOCUMENT	LITERAL	BARE	В операциях WS допускается только один параметр, который связан с корневым элементом схемы и который полностью определяет содержимое сообщения
DOCUMENT	LITERAL	WRAPPED	Все параметры операции WS оборачиваются в корневой элемент схемы с тем же самым именем, что и операция, к которой он относится.

Аннотация @WebParam

@javax.jws.WebParam используется для настройки WSDL-разметки, генерируемой для @WebParam - методов. Аннотация применяется к параметрам методов и содержит следующие атрибуты:

- name – в режиме RPC/LITERAL задает своим значением xxx WSDL-элемент <part name=”xxx” …>. В остальных режимах этот атрибут задает внутри документа XML-схемы локальное XML-имя элемента, связанного с аннотируемым параметром. Если атрибут не задан, то по-умолчанию используется значение “argN”, где N – порядковый номер с 0 аннотируемого параметра в сигнатуре метода.
- header – атрибут, который используется для обозначения того, куда параметр будет помещен: в SOAP-заголовок или в SOAP-тело. По-умолчанию = false = тело. В соответствующем элементе <binding><operation><input> WSDL-документа, соответственно, будут возникать <soap:header message=… part=”xxx”> и <soap:body parts...>.
- partName – атрибут, управляющий созданием свойства name в WSDL-элементе <part> или схемой XML-параметра для стилей RPC и DOCUMENT/BARE.
- targetNamespace – атрибут, использующийся либо для стиля WS DOCUMENT/LITERAL, либо в случае, когда аннотируемый параметр помечен атрибутом header. Его значение задает целевое пространство имен (targetNamespace) описания схемы, содержащей элемент параметра.
- mode = {IN, OUT, INOUT} – атрибут, задающий способ использования параметра:

прием от WS, передача к WS, или и то и другое. По-умолчанию = IN. При использовании параметра как выходного (OUT, INOUT), его необходимо заключать в специальную оболочку javax.xml.ws.Holder для обеспечения передачи объекта параметра «по ссылке», т.к. WS не поддерживает состояние (?).

Ex : пример аннотирования параметров для JAX-WS

```
@WebMethod(operationName = "CheckStatus")
public int checkStatus(
  @WebParam(name = "ReservationID")
  String reservationId,
  @WebParam(name = "CustomerID", mode = WebParam.Mode.OUT)
  javax.xml.ws.Holder<Integer> customerId
){
  ...
  // возврат id заказчика и статуса
  customerId.value = getCustomerId(reservationId);
  return status;
}
```

Ex : пример сгенерированного компилятором JAX-WS фрагмента WSDL-документа с описанием параметров операции

```
<message name="CheckStatus">
  <part name="ReservationID" type="xsd:string"/>
  <part name="CustomerID" type="xsd:int"/>
</message>
<message name="CheckStatusResponse">
  <part name="return" type="xsd:int"/>
</message>
<portType name="TravelAgent">
  <operation name="CheckStatus" parameterOrder="ReservationID CustomerID">
    <input message="tns:CheckStatus"/>
    <output message="tns:CheckStatusResponse"/>
  </operation>
</portType>
```

Аннотация @WebResult

@javax.jws.WebResult является несколько упрощенным аналогом @WebParam для возвращаемого методом результата. Применяется к методам и имеет следующие атрибуты (аналогичные по смыслу атрибутам @WebParam):

- name – задает имя результата операции в WSDL-документе. По-умолчанию для стиля DOCUMENT/LITERAL/BARE это <имя_операции_WSDL> + “Response”. Для остальных стилей это константа ”result”.
- targetNamespace – задает целевое пространство имен схемы, содержащей описание элемента результата для стиля DOCUMENT

- partName — определяет имя возвращаемого значения аналогично @WebParam
- header — определяет место размещения результата — в теле (false) или заголовке (true) сообщения

Ex : пример аннотирования результата метода для JAX-WS

```
package com.titan.webservice;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebResult;
@Stateless
@WebService(name = "TravelAgent")
public class TravelAgentBean
{
  @WebMethod(operationName = "Reserve")
  @WebResult(name = "ReservationID")
  public String makeReservation(int cruiseId, int cabinId, int customerId, double price) {
  ...
  }
}
```

Ex : пример фрагментов WSDL-документа с описанием результата операции, сгенерированного компилятором JAX-WS

```
<xs:element name="ReserveResponse" type="ReserveResponse"/>
<xs:complexType name="ReserveResponse">
 <xs:sequence>
   <xs:element name="ReservationID" type="xs:string" nillable="true"/>
 </xs:sequence>
</xs:complexType>
...
<message name="ReserveResponse">
  <part name="parameters" element="tns:ReserveResponse"/>
</message>
...
<portType name="TravelAgent">
  <operation name="Reserve">
    <input message="tns:Reserve"/>
    <output message="tns:ReserveResponse"/>
  </operation>
</portType>
```

Аннотация @OneWay и асинхронные вызовы WS

@javax.jws.OneWay используется для определения операции WS, которая будет возвращать пустое сообщение. В соответствующем такой операции элементе <operation> будет отсутствовать элемент <output>. Это позволяет в теории обращаться к WS асинхронно. Будет ли дергаться метод асинхронно на практике, зависит от

серверной реализации JAX-WS.

Rem: настоящие асинхронные запросы реализуются клиентским прокси при помощи специальных методов SEI типа Response<T> invokeAsync(T message) и java.util.concurrent.Future<?> invokeAsync(T msg, AsyncHandler<T> handler). Последний метод требует реализации обработчика AsyncHandler. Для автогенерации в SEI подобных методов в WSDL-документе надо прописать <jaxws:bindings><enableAsyncMapping>true</enableAsyncMapping>...</jaxws:bindings> соответствующего элемента <wsdl:portType>.

Аннотация @HandlerChain и обработчики сообщений

@javax.jws.HandlerChain используется для определения набора обработчиков сообщений SOAP, которые должны вызываться в ответ на получение сообщения. С логической точки зрения это аналоги перехватчиков EJB. Обработчики бывают 2 типов: логическими (@javax.xml.ws.handler.LogicalHandler) и протокольными (@javax.xml.ws.handler.SOAPHandler). Первые перехватывают полезную нагрузку, вторые все элементы SOAP-сообщений.

Ниже пример программного включения обработчика SOAPHandler в клиентский Service-класс, перехватывающего исходящий запрос и добавляющего в его SOAP-заголовок учетную информацию:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" ...>
<env:Header>
<ns1:Security ns2:ID="simple" xmlns:ns1= "http://ns1.xsd" xmlns:ns2="http://ns2.xsd">
  <ns1:UserName>User Name</ns1:UserName>
  <ns1:Password>12345</ns1:Password>
</ns1:Security>
</env:Header>
<env:Body>…</env:Body>
</env:Envelope>
```

```
// Код включения обработчика в Service-класс
…
ExService exService = new ExService();
exService.setHandlerResolver(new ExHandlerResolver());
…
// Определение оболочки обработчиков и вложенного обработчика
private class ExHandlerResolver implements HandlerResolver {
// Переопределение списка обработчиков
  @Override
  public List<Handler> getHandlerChain(PortInfo portInfo) {
    List<Handler> handlerList = new ArrayList<>();
    handlerList.add(securityHandler);
    return handlerList;
  }
```

```
// Класс — обработчик сообщения SOAP
  private final SOAPHandler securityHandler = new SOAPHandler<SOAPMessageContext>() {
// Метод, модифицирующий заголовок SOAP добавкой элементов с учетной информацией
    @Override
    public boolean handleMessage(SOAPMessageContext ctx) {
      Boolean outboundProperty = (Boolean)
ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
// Исходящий запрос
        if (outboundProperty) {
          try {
            String prefix1 = "ns1";
            String prefix2 = "ns2";
            String nsURI1 = "http://ns1.xsd";
            String nsURI2 = "http://ns2.xsd";
            SOAPFactory factory = SOAPFactory.newInstance();
            SOAPElement security = factory.createElement("Security", prefix1, nsURI1);
            security.addAttribute(new QName(nsURI2, "ID", prefix2), "simple");
            SOAPElement username = security.addChildElement("Username", prefix1, nsURI1);
            username.setTextContent("User Name");
            SOAPElement password = security.addChildElement("Password", prefix1, nsURI1);
            password.setTextContent("12345");
            SOAPEnvelope envelope = context.getMessage().getSOAPPart().getEnvelope();
            SOAPHeader header = envelope.getHeader();
            header.addChildElement(security);
          } catch (SOAPException ex) {
          }
        }
        return true;
      }
    }
}
```

Аннотация @WebServiceProvider

Аннотация @javax.xml.ws.WebServiceProvider является аналогом @WebService и определяет SEIC. Но в отличие от @WebService она снимает с JAX-WS ответственность за разбор входящего SOAP-сообщения и формирование исходящего и возлагает ее на сам SEIC. Это позволяет обойтись без прокси-заглушек и даже WSDL, т.е. создавать ориентированные на XML-сообщения WS, в т.ч. RESTful WS. @WebServiceProvider SEIC должен реализовывать интерфейс Provider<T>. В дополнение используются @BindingType и @ServiceMode. При использовании @BindingType(value=HTTPBinding.HTTP_BINDING) SEIC представляет RESTful WS.

Отделение контракта компонента JAX-WS в отдельный SEI

Альтернативой к описанию компонента JAX-WS в SEIC является вынос метаинформации в отдельный интерфейс (SEI). Его следует определить в атрибуте endpointInterface аннотации @WebService SEIC. Этот внешний интерфейс представляет

все свои методы как операции веб-службы. Он помечается единственно аннотацией @WebService. Сам SEIC не обязан реализовывать связанный с нею SEI. Но с другой стороны никто этого не запрещает, тут как удобней.

```
Ex : пример определения отдельного SEI для JAX-WS

package com.titan.webservice;
import javax.jws.WebService;
@WebService
public interface TravelAgent{
  public java.lang.String makeReservation(int cruiseId, int cabinId, int customerId, double price);
}
```

Реализующий веб-сервис SEIC затем привяжет этот интерфейс при помощи своей аннотации @WebService:

```
Ex : привязывание отдельного SEI к Endpoint EJB для JAX-WS

package com.titan.webservice;
import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService(endpointInterface = "com.titan.webservice.TravelAgent")
public class TravelAgentBean implements TravelAgent {
...
}
```

Клиентская сторона. Класс службы (Service Class)

В JAX-WS существует класс при помощи которого клиент может получить заглушки для связи с веб-службой. Этот класс должен расширять javax.xml.ws.Service и обеспечивать метод получения SEI.

Вместе с данным классом используется @javax.xml.ws.WebServiceClient для определения имени Endpoint WS (<service name=”ProcessorService”> WSDL), пространства имен WSDL и местоположения WSDL-документа требуемого WS.

Аннотация @javax.xml.ws.WebEndpoint используется для определения конкретного SEI (через <service><port name=”ProcessorPort”> WSDL), возвращаемого прокси.

```
Ex : пример определения Service Class JAX-WS, загружающего WSDL-файл с сервера

package com.charge_it;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebEndpoint;

@WebServiceClient(name="ProcessorService", targetNamespace="http://charge-it.com/Processor",
```

```
   wsdlLocation="http://charge-it.com/Processor?wsdl")
public class ProcessorService extends javax.xml.ws.Service {
  public ProcessorService( ) {
    super(new URL("http://charge-it.com/Processor?wsdl"),
         new QName("http://charge-it.com/Processor", "ProcessorService"));
  }
  public ProcessorService(String wsdlLocation, QName serviceName) {
    super(wsdlLocation, serviceName);
  }
  @WebEndpoint(name = "ProcessorPort")
  public Processor getProcessorPort( ) {
    return (Processor)
      super.getPort(new QName("http://charge-it.com/Processor", "ProcessorPort"),Processor.class);
  }
}
```

Пример Service-класса автономного JSE-клиента:

```
Ex.: пример Service Class JSE-клиента с загрузкой WSDL из локального файла

@WebServiceClient(targetNamespace = "http://jaxws/")
public class SEIService extends javax.xml.ws.Service {
// Загрузка из файла SEIC.wsdl, лежащего недалеко от файла SEI.class
  public SEIService() throws Exception {
    super(SEI.class.getResource("../../META-INF/jaxws/SEIC.wsdl"), new
QName("http://jaxws/","SEICPort"));
  }
  public SEIService(URL wsdlURL) throws Exception {
    super(wsdlURL, new QName("http://jaxws/","SEIC"));
  }
  @WebEndpoint(name="SEI")
  public SEI getSEIPort(){
    return super.getPort(new QName("http://jaxws/","SEICPort"),SEI.class);
  }
}
```

Клиентская сторона. Интерфейс конечной точки WS (SEI)

Клиент может использовать в качестве SEI тот же самый отдельный SEI, что и на стороне сервера, те же самые аннотации. Стоит отметить, что JAX-WS поддерживается в JSE, поэтому есть возможность создавать для клиента эти интерфейсы самостоятельно ручками.

```
Ex : пример клиентского SEI на платформе JEE

package com.charge_it;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
```

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface Processor{
  public int charge(String name, String number, java.util.Calendar expDate, String cardType, float
amount);
}
```

Клиентская сторона. Аннотация @WebServiceRef

JAX-WS определяет аннотацию @javax.xml.ws.WebServiceRef для внедрения в клиента напрямую либо SEI-заглушки, либо Service-класса. Аннотация применима к полю, методу, типу. Она содержит следующие атрибуты:

- name – определяет как веб-служба будет связана с ENC JNDI
- wsdlLocation – URL по которому живет WSDL-документ, локальный или удаленный, неважно. По-умолчанию значение этого атрибута извлекается из Service Class, который был аннотирован @WebServiceClient
- mappedName – вендорный глобальный ID для WS. Подробности нужно узнавать в вендорной документации.
- value, type – используются для внедрения в свойство как Service-класса, так и SEI. Для того чтобы внедрить SEI, надо указать в свойстве value тип Service. Тогда в type можно указать тип SEI, который и будет внедрен. Для внедрения Service-класса надо и в type и в value задать тип Service. Атрибуты value и type можно не определять, если EJB-контейнер сможет вывести соответствующую информацию из свойства, в которое идет внедрение.

Пример внедрения в поле endpoint ссылки на SEI Processor с указанием в value Service-класса и автовыводом (type опущен) типа SEI из типа поля внедрения:

```
@WebServiceRef(ProcessorService.class)
private Processor endpoint;
```

Пример внедрения в поле service ссылки на Service-класс ProcessorService с автовыводом типа Service (value и type опущены) из типа поля внедрения:

```
@WebServiceRef
private ProcessorService service;
```

Ex : пример внедрения ссылки на SEI Processor в поле processor класса SFSB с указанием EJB-контейнеру класса веб-службы ProcessorService

```
package com.titan.travelagent;
import com.charge_it.ProcessorService;
import com.charge_it.Processor;
...
@Stateful
```

```
public class TravelAgentBean implements TravelAgentRemote {
    ...
    @WebServiceRef(ProcessorService.class)
    Processor processor;
     ...
    public TicketDO bookPassage(CreditCardDO card, double price) ... {
      ...
      processor.charge(customerName, card.number, expDate, card.type, price);
    }
    ...
}
```

В этом примере аннотация заставляет внедрять в поле processor динамический прокси, который реализовывает интерфейс конечной точки Processor. Поскольку тут производится прямое внедрение SEI в поле, то требуется указать в @WebServiceRef атрибут value = ProcessorService.class, который будет задействован в создании заглушки, реализующей этот SEI.

Rem: в Glassfish можно переопределить атрибуты @WebServiceRef в элементе <service-ref> вендор-специфических XML-дескрипторов glassfish-.xml*

Особенности клиента в JEE. Вызов WS при помощи CDI

Спецификация JAX-WS в JEE7 не интегрировала SOAP веб-службы в технологию CDI (Context and Dependency Injection). В результате клиент не может напрямую внедрить заглушки и классы службы JAX-WS в свой CDI-управляемый компонент при помощи @Inject.

Для решения этой проблемы можно применять костыль — CDI-управляемый класс-производитель CDI:

```
// CDI-управляемый класс-производитель
public class WSProducer {
  @Produces
  @WebServiceRef
  private ProcessorService service;
}

//CDI-внедрение класса службы ProcessorService через класс-производитель  WSProducer
// JEE-движок найдет производителя по типу  ProcessorService и @Produces
@Stateful
public class TravelAgentBean implements TravelAgentRemote {
    @Inject
    private ProcessorService service;
    public TicketDO bookPassage(CreditCardDO card, double price) ... {
      Processor processor = service.getProcessorPort();
      processor.charge(customerName, card.number, expDate, card.type, price);
      ...
    }
```

```
}
```

Rem: в CDI-управляемые компоненты попадают все POJO, содержащиеся в war или jar архивах, содержащих дескриптор beans.xml. Туда не могут попасть примитивы, стандартные классы вроде String, упакованные в приложении в отдельный архив rt.jar, или классы, созданные чужими фреймворками, в посторонних архивах. Для разрешения этой проблемы создан костыль в форме вспомогательно CDI-управляемого класса-производителя, который по сути является оберткой для проблемных классов и примитивов, которые необходимо внедрить. Внутри этой обертки поля требуемых типов помечаются @Produces и, если надо, аннотацией-меткой. Кроме полей, так же можно помечать и методы, которые превращаются в фабрики экземпляров типа. После чего внедрение производится стандартной @Inject. При этом работает автоматическое распознавание типов, что позволяет не использовать аннотацию-метку для уникальных внедряемых типов или использовать одну такую метку для набора различных типов.

Обработка исключений

При возникновении сбоев в обработке запросов, WS должен уведомить об этом потребителя. Для этого в ответном сообщении используется элемент <Fault> SOAP .

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>java.lang.NullPointerException</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

При выбросе из метода исключения, движок JAX-WS по-умолчанию автоматически заполняет значение элемента <faultstring> F.Q.N. исключения, а также <faultcode> одним из кодов (Server, Client, ...), определенных в пространстве имен SOAP (в примере — soap:Server, что подразумевает сбой на сервере WS).

Rem: элементы <faultcode> и <faultstring> принадлежат локальным пространствам имен (беспрефиксным) (?).

В частности, исключения можно использовать в бизнес-логике метода и выбрасывать программно (throw), используя механизм JAX-WS, оборачивающий их в <Fault> , для сообщению потребителю причины невозможности обработки запроса. Выбрасывать можно любые непроверяемые (RuntimeException) или проверяемые (все остальные Exception исключая RuntimeException). В частности, можно использовать непроверяемое исключение javax.xml.ws.WebServiceException и его наследников.

Можно создать собственное исключение и настроить его передачу при помощи аннотации JAX-WS @WebFault:

```
@WebFault(name=”ExFault”)
public class ExException extends Exception {
  public ExException() {
    super();
  }
  public ExException(String message) {
    super(message);
  }
}

@WebService
public class ExService {
  public String exMethod(String id) throws ExException {
    if(!validate(id)) {
      throw new ExException(“Bad id”);
    }
    ...
  }
}
```

В данном примере создано собственное проверяемое исключение ExException, в SEIC метод exMethod выбрасывает это исключение, если входной id неправильный. В ответ на неправильный id в SOAP-ответе появится элемент <Fault> с дополнительным описанием:

```
<soap:Envelope xmlns:soap=”http://schemas.xmlsoap.org/soap/envelope/”>
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>org.example.ExException</faultstring>
      <detail>
        <ns2:ExFault xmlns:ns2=”http://example.org/”>
          <message>Bad id</message>
        </ns2:ExFault>
      </detail>
    <soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Для программного создания исключений с заданием своего faultcode можно использовать непроверяемые исключения класса javax.xml.soap.SOAPFaultException (наследника WebServiceException). Создавать их удобно при помощи javax.xml.soap.SOAPFactory:

```
@WebService
public class ExService {
  public String exMethod(String id) throws SOAPFaultException {
    if(!validate(id)) {
      SOAPFactory sf = SOAPFactory.newInstance();
      SOAPFault fault = sf.createFault(“Bad id”, new Qname(“ExFault”));
```

```
    throw new SOAPFaultException(fault);
   }
   ...
  }
}
```

В результате выброса исключения потребитель получит следующее сообщение:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <soap:Fault>
   <faultcode>ExFault</faultcode>
   <faultstring>Bad id</faultstring>
  </soap:Fault>
 </soap:Body>
</soap:Envelope>
```

Жизненный цикл конечной точки WS & Callback

Поведение JAX-WS Endpoint (SEIC) в JSE (JAX-WS RI) ничем не отличается от поведения POJO.

В JEE любой JAX-WS Endpoint - и EJB Endpoint и Servlet Endpoint являются компонентами без сохранения состояния, со стандартным жизненным циклом и поддержкой обратных вызовов:

Not exists – @PostConstruct – Ready – Вызов метода – @Predestroy – Not exists

Контейнер вызывает @PostConstruct callback после создания экземпляра WS и @PreDestroy callback при его ликвидации, если эти методы обратного вызова существуют, конечно.

В EJB-модели в жизненный цикл могут вмешиваться перехватчики.

WebServiceContext

SOAP WS обладают контекстом среды javax.xml.ws.WebServiceContext, объект которого можно получить при помощи стандартного для JSE внедрения @Resource. Связанный с этим контекстом SEIC может получить из него в run-time информацию о текущих сообщении и состоянии компонента веб-службы в целом:

- getMessageContext – получение MessageContext, контекста сообщения SAAJ, из которого можно получить информацию о текущем запросе, получить доступ к заголовку и телу SOAP и т.д.
- getUserPrincipal – получение Principal, идентифицирующего клиента, сделавшего запрос.
- isUserInRole – true/false принадлежности аутентифицированного клиента

указанной логической роли

- getEndpointReference – получение EndpointReference, содержащей инфорамацию, связанную с собственной реализацией (ссылки на заданный SEIC, URL доступа, URL WSDL-документа, ENC JNDI - имена SEI/SEIC и Service-класса).

```
Ex.: SEIC

@WebService
public class ExWS {
  @Resource
  private WebServiceContext context;
…
  @WebMethod
  public boolean validate(String id) {
    if (!context.isUserInRole("Admin"))
      throw new SecurityException("Validate failed: administrator permission required!");
    }
…
  @WebMethod
  public W3CEndpointReference getExWSEndpointReference(){
    return (W3CEndpointReference) context.getEndpointReference(null);
  }
…
}
```

В примере выше в Endpoint Servlet (Web-модель) внедряется WebServiceContext, который используется в методе validate для авторизации пользователя (должен соответствовать роли “Admin”). Метод getExWSEndpointReference выплевывает расширенный W3CEndpointReference с описанием сообщения и состояния всей службы.

```
Ex.: логгер LogEJB и клиент WS - ExServlet

@Stateless
public class LogEJB {
  public void log(W3CEndpointReference ref) {
    ExWS port = ref.getPort(ExWS.class);
    System.out.println("#### W3CEndpointReference info ####");
    System.out.println(ref);
  }
}

@WebServlet(name = "ExServlet", urlPatterns = {"/ExServlet"})
public class ExServlet extends HttpServlet {
// Внедрение Service-класса WS
    @WebServiceRef(wsdlLocation="http://Localhost:8080/ExWS")
    private ExWSService service;
    @EJB
    LogEJB logger;
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
```

```
IOException, ServletException {
…
  W3CEndpointReference ref = service.getExWS().getExWSEndpointReference ();
  logger.log(ref);
…
  }
}

Вывод:
#### W3CEndpointReference info ####
…
<EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
  <Address>http://Localhost:8080/ExWS</Address>
  <Metadata wsdli:wsdlLocation="http://ex.ws/ http://Localhost:8080/ExWS?wsdl">
    <wsam:InterfaceName xmlns:wsns="http://ex.ws/" >wsns:ExWS</wsam:InterfaceName>
    <wsam:ServiceName EndpointName="ExWS" xmlns:wsns="http://ex.ws/" >
      wsns: ExWSService
    </wsam:ServiceName>
  </Metadata>
</EndpointReference>
…
```

В примере выше определен логгирующий SLSB LogEJB, который выводит в консоль информацию о состоянии порта (SEIC ExWS) веб-службы по параметру W3CEndpointReference (расширяющему EndpointReference). Этот LogEJB вызывает клиент — сервлет ExServlet при каждом запросе и выводит на консоль информацию о конечной точке: URL доступа, URL WSDL-документа, имя WSDL-порта (SEI), имя Service-класса.

Упаковка веб-служб JAX-WS

Для JSE (JAX-WS RI) все упаковывается в стандартный jar-архив. В JEE веб-службы JAX-WS могут быть упакованы в jar-архив в модели EJB или в war-архив в Web-модели. В архиве должно содержаться следующее:

- реализация компонента WS, т.е. SEIC с зависимостями
- опционально SEI
- WSDL-документ либо файлом, либо ссылкой. При использовании аннотаций WSDL необязателен, т.к. они полностью его заменяют.
- опционально XML-схемы для сообщений SOAP (WSDL-элемент <types>), JAX-WS также проводит их автогенерацию
- опционально дескриптор развертывания webservices.xml (для JSE не описан (?))

Публикация веб-службы JAX-WS

Веб-службы JAX-WS могут быть развернуты простым JSE-приложением, поскольку JAX-WS включен в JSE6 и выше. Для этого существует эталонная реализация JAX-WS RI, которую можно использовать для публикации. JAX-WS RI использует легкий HTTP-

сервер из состава Oracle JVM (com.sun.net.httpserver), а публикация WS проводится при помощи статического метода javax.xml.ws.Endpoint.publish:

```
Ex: пример самозапускающейся JAX-WS WS

@WebService(targetNamespace = "http://jaxws/")
public class ExService {
  @WebMethod(operationName = "exMethod")
  public Item exMethod(String param){ … }
…
  public static void main(String[] args) {
    Endpoint.publish("http://localhost:8080/exService",  new ExService());
  }
}
```

После выполнения этого кода, по адресу http://localhost:8080/exService поднимется веб-служба ExService, готовая принимать SOAP-запросы.

Rem: применение логических имен вроде operationName позволяет избавиться от привязывания алиасов к F.Q.N. реализации

Также JAX-WS веб-службу можно поднять на любом JEE7 сервере приложений вроде Glassfish. И также можно использовать для публикации в Web-модели скрутку Web-сервера вроде Tomcat или Jetty с отдельной реализацией JAX-WS вроде CXF, Axis2, Metro.

Другие аннотации и API

JAX-WS определяет также еще некоторые аннотации для дальнейшего развития и наборы клиентских API для динамического вызова веб-служб и поддержки некоторых новейших WS-* спецификаций. Информацию по ним можно найти в спецификациях и специальной литературе).

RESTful WS

REST (Representational State Transfer, передача репрезентативного состояния) – концепция (архитектура) взаимодействия компонентов распределенного приложения, введеннуя одним из разработчиков HTTP (Roy Fielding). Системы, использующие данную архитектуру называются RESTful-системами.

Клиент RESIful WS обращается к ресурсам веб-приложения и может получить в ответ одно из их представлений. Ресурс RESTful WS – любое нечто, которое клиент может запросить у веб-службы по ссылке или с которым он может взаимодействовать. Т.е. это все, что заслуживает отдельной ссылки. RESTful WS должна быть адресуемой, т.е. представлять в виде ресурсов все значимые информационные фрагменты приложения, реализующего ее. Также REST предписывает реализации RESTful WS быть как можно

более сильно-связанной (в смысле орграфов это значит, что все вершины связаны между собой простыми путями). Т.е. снабжать логически-связанные ресурсы прямыми ссылками, что позволяет клиенту получить доступ ко всем ресурсам веб-службы по единственной входной ссылке. Сильно-связанные RESTful приложения называют гипермедийными: представление каждого ресурса содержит помимо прочего информацию о навигации и доступе; каждый переход на новый ресурс меняет состояние приложения.

Представление ресурса — любая информация об его состоянии. Различают человеко-читаемые (XML) и машинно-обрабатываемые представления (JSON). Репрезентативность состояния означает, что одно и то же состояние ресурса может быть передано в различных представлениях.

Для получения ресурсов используются операции веб-службы. Если SOAP WS использует протокол HTTP лишь в качестве транспорта из-за требований мультипротокольности , то RESTful WS использует его на полную катушку, как прикладной, реализуя для работы с ресурсами базовые HTTP-операции (GET, POST, DELETE, PUT, HEADER, OPTIONS), дающие CRUD-функциональность (Create, Read, Update, Delete). В отличие от SOAP, где сообщение определяет операцию, в REST каждый ресурс и операция определяются уникальным URI, параметры идут в запросе, а в теле сообщения может быть какая-угодно информация (текст, XML, BLOB и т.д.). Представление ресурса RESTful также может задаваться отдельным URI, но может и определяться содержимым запроса.

В JEE RESTful WS описывается фреймворком JAX-RS. Аналогично JAX-WS, экспортировать в виде JAX-RS компонента можно любой POJO. Экспорт EJB позволяет использовать все преимущества EJB-контейнера.

HTTP и REST

REST построен на прикладном протоколе HTTP. Запрос HTTP представляет собой набор строк, разделенных символом конца строки. Состоит он в общем случае из заголовка и тела, разделенными пустой строкой (двойным символом конца строки). Вид запроса зависит от типа операции. Например, GET-запрос состоит из одного заголовка.

```
Ex: GET-запрос HTTP

GET http://localhost/res?param1=1&param2=2#section1 HTTP/1.1
Host: localhost
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: ru
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
Proxy-Connection: Keep-Alive
```

В первой строчке заголовка идут через пробелы тип метода, URL (или его часть с параметрами) запрашиваемого документа, тип протокола. В остальных строках

перечислены параметры заголовка (заголовки) в формате <имя_заголовка>:<значение>. Обязательный заголовок Host указывает доменное имя или IP-адрес сервера (хоста). Остальные заголовки несут техническую информацию, ниже перечислена часть наиболее употребимых:

HTTP-заголовок	**Описание**
Accept	Допустимые типы содержимого ответа (например, text/plain)
Accept-Charset	Допустимые наборы символов ответа (например, utf-8)
Accept-Encoding	Допустимые варианты кодировки ответа (например, gzip, deflate)
Accept-Language	Допустимый язык отклика ответа (en-US)
Cookie	Файл HTTP-cookie, ранее отосланный сервером
Content-Length	Длина тела запроса в байтах
Content-Type	MIME-тип тела сообщения (например, text/xml)
Date	Дата и время отправки сообщения
ETag	Идентификатор конкретной версии ресурса (например, 8af7ad3082f20958)
If-Match	Действие производится лишь в том случае, если клиент предоставил объект, совпадающий с таким же объектом на сервере
If-Modified-Since	Допускает возврат кода состояния 304 — Не изменялось, если контент не изменялся с указанной даты
User-Agent	Строка, соответствующая пользовательскому агенту (например, Mozilla/5.0)

URL (Uniform Resource Locator) – частный случай URI (Universal Resource Identifier), использующийся в HTTP-запросах для определения местонахождения документа. Он имеет вид: <схема>://<хост>:<порт>/<URLпуть>?<параметры>#<якорь>

POST-запрос содержит помимо HTTP-заголовка и тело, отделенное пустой строкой. В URL POST-запроса параметры передавать не принято (не факт, что сервер будет их искать там), они записываются в теле.

```
Ex: POST-запрос HTTP

POST http://localhost/ HTTP/1.0
Host:  localhost
Referer: http://localhost/index.html
Cookie: income=1
Content-Type: application/x-www-form-urlencoded
Content-Length: 35

login=User&password=12345
```

Rem: в NetBeans используется сниффер Apache tcpmon, который можно применять для анализа входных и выходных HTTP-сообщений . Также для тестирования можно применять консольную утилиту cURL (http://curl.haxx.se/), позволяющую отправлять из консоли HTTP-запросы и выводить на консоль ответы .

В REST используется 2 понятия, связанные с взаимозависимостью операций при обращении к ресурсам:

- идемпотентность - означает, что повторное применение операции будет проигнорировано. Т.е. AA=A, где A – оператор операции, воздействующий на состояние системы. Идемпотентными должны быть операции GET, DELETE, PUT, HEAD, OPTIONS
- безопасность — означает, что операция не меняет состояния системы. Т.е. Ax=x, где x – состояние системы, а A – оператор операции. Безопасными являются операции чтения GET, HEAD, OPTIONS, что позволяет использовать их асинхронно.

В RESTful WS тип запроса HTTP определяет порядок его обработки и операцию WS. Из существующих 8 типов для определения операций используются 6.

- GET – чтение данных, не меняющее состояния сервера (Ax = x)
- DELETE – операция по удалению ресурса, ее повтор игнорируется (AA = A)
- POST – изменение данных на сервере. Запрос может содержать данные для изменения, а может и не содержать. Ответ также может содержать, а может и нет данные результата. При каждом повторном запросе может создаваться новый независимый объект, а в ответ выдаваться его уникальный URI (AA!=A, Ax=y)
- PUT – сохранение или обновление на сервере данных, переданных в теле запроса (AA=A)
- HEAD – аналогичен GET, но возвращает только код ответа и заголовки, связанные с запросом, без тела. Удобен для валидации ссылок, получения информации о длине ответа и других метаданных (AA=A)
- OPTIONS – возвращает информацию о вариантах взаимодействия в цепи запрос-ответ для данного URI.

Rem: для GET-запросов в браузерах обычно стоит ограничение в 4-8 KByte.

Ответ также содержит заголовок и, возможно, тело. Конкретное представление ответа может выбираться согласованием по содержанию запроса. Это механизм автоматического выбора веб-приложением представления среди множества допустимых по специальным заголовкам запроса (Accept, Accept-*, User-Agent).

Представление ответа и формат тела запроса определяется MIME-типом . Значения MIME-типов записываются в заголовки Accept и Content-Type HTTP-сообщения, соответственно. Эта информация используется для последующей типизации принимающей стороной. MIME-типы считаются открытыми для расширения (?). В HTTP применяется 5 категорий MIME-типов (text, image, audio, video, application). Каждый MIME тип кодируется категорией и именем, например:

- text/plain; text/html; text/xml

- image/gif; image/jpeg; image/png
- application/xml; application/json

Коды состояния HTTP передаются в ответ (в первой строке) и определены в спеке RFC 2616 (HTTP/1.1). Каждый код состоит из 3 цифр nnn, первая цифра означает категорию кода:

- 1xx – запрос обрабатывается
- 2xx – запрос успешно обработан
- 3xx – перенаправление (нужно сделать перезапрос)
- 4xx – клиентская ошибка
- 5xx – серверная ошибка

```
Ex: пример успешного ответа (200) на HTTP-запрос

HTTP/1.1 200 OK
Date: Mon, 07 Apr 2003 14:40:25 GMT
Server: Apache/1.3.20 (Win32) PHP/4.3.0
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/plane

… Message …
```

Описание JAX-RS

Сам REST – архитектурный стиль или шаблон проектирования, а не стандарт, в отличие от SOAP WS и WS-I. Это лишь пожелания, выполнение которых делает веб-службу RESTful WS согласно RFC 2616.

В принципе реализовать RESRful WS в JEE можно и самопальным сервлетным приложением. Но код такой службы будет оригинальным и плохо читаемым. Для автоматизации создания RESTful WS и был придуман JAX-RS, включенный в JEE начиная с 6 версии. Это просто еще один удобный способ использования HTTP-протокола без необходимости реализации javax.servlet.Servlet и позволяющий вызывать методы напрямую, сразу передавая им параметры.

В JEE7 используется JAX-RS 2.0. Этот интерфейс определяет серверный и клиентский API для построения и использования RESTful WS, фильтры и перехватчики сообщений (аналогично SOAP Handler и EJB Intercepter), асинхронный режим для длительных операций и оповещений со стороны сервера, интеграцию с технологией валидации компонентов. JAX-RS 2.0 не был реализован в JSE, в отличие от JAX-WS 2.0, но благодаря простоте получил множество вендорных реализаций на основе уже существующего Java API: эталонная реализация Jersey (Glassfish), Apache CXF и другие.

JAX-RS WS строится на базе обычных POJO (в т.ч. EJB) при помощи аннотаций. При этом стандартно для JEE-компонентов, компоненты JAX-RS снабжаются поведением по-умолчанию, которое донастраивается при помощи метаинформации. Специального XML-дескриптора спецификацией не предусмотрено.

JAX-RS состоит из следующих пакетов:

- javax.ws.rs – интерфейсы и аннотации для построения высокоуровнего построения RESTful WS
- javax.ws.rs.client – классы и интерфейсы для построения клиента JAX-RS 2.0
- javax.ws.rs.core – низкоуровневый API для построения классов ресурсов JAX-RS
- javax.ws.rs.ext – расширение типов, используемые в JAX-RS
- javax.ws.rs.container – контейнерный API (?)

Преимущества и область применения JAX-RS / RESTful WS

Преимуществом RESTful WS является простота. Большинство платформ имеют все необходимое сразу и не требуют дополнительных библиотек, как в случае SOAP WS. К тому же в силу простоты и универсальности REST дает лучшую межоперабельность по сравнению с SOAP. SOAP WS несмотря на профили WS-I имеет проблемы с совместимостью вендор-специфических реализаций (проблема с .NET WS).

Другим преимуществом REST является компактность запроса вместе с произвольностью типа ответа, в т.ч. JSON, обрабатываемого HTML5/WEB2.0 – браузерами напрямую. Для сравнения, SOAP запросы и ответы имеют XML-структуру, относительно сложную в обработке и достаточно дорогую для транспорта из-за большой доли метаинформации. REST дает больше возможностей по оптимизации запросов и ответов, что особенно важно для мобильных платформ (в IOS встраивается REST, а не SOAP).

Когда выгодно экспортировать EJB как JAX-RS WS

Главным функциональным отличием EJB от POJO является поддержка транзакций и окружения безопасности. Если они нужны в WS, то имеет смысл создавать компоненты WS на основе EJB.

При экспорте EJB есть некоторые ограничения и проблемы:

- во-первых RESTful WS не должны поддерживать клиентское состояние между запросами. Поэтому для экспорта подходят лишь SLSB и @Singleton, однако у последних проблемы с параллелизмом и в качестве компонентов JAX-RS WS используют лишь SLSB.
- во-вторых RESTful WS по сути надстройка над прикладным протоколом HTTP, а

это означает, что Java-компонент такой сетеобразной веб-службы должен подстраиваться под посторонний протокол, например, выдавать JSON вместо стандартного java.util.List. Для разгрузки EJB-слоя от подобной специфической поддержки надо использовать классы-адаптеры. Впрочем JAX-RS это значительно автоматизировал.

- в-третьих RESTful WS предназначены для обработки простых запросов, т.е. операции должны быть достаточно раздроблены для того, чтобы содержать параметры простых типов. Операции со сложными типами лучше оформлять в виде JAX-WS, т.к. несмотря на возможность их XML-маршалинга в JAX-RS посредством JAXB, в SOAP это лучше автоматизированно благодаря дополнительному использованию XSD и WSDL.

Применение JAX-RS

Веб-служба создается реализацией JAX-RS как сервлетное приложение. Для ее организации используются аннотации, вспомогательные классы поставщиков, конфигурационный класс, собирающий ресурсы в единую веб-службу. Специальных XML-дескрипторов спекой не предусмотрено. Для преобразования POJO в компонент JAX-RS (ресурс) достаточно всего 2-х аннотаций: @javax.ws.rs.Path и @javax.ws.rs.GET/POST/PUT/DELETE

Параметры запросов к RESTful WS встраиваются простой строкой либо в URL заголовка, либо в один из других заголовков, либо в тело HTTP-сообщения, вместо XML-структуры SOAP WS. В теле HTTP-запроса может быть вложение, извлекаемое WS как MIME-тип. Возвращаемым значением может быть что угодно, в т.ч. и данные в формате JSON, обрабатываемые современными браузерами (HTML 5.0), значительно быстрее SOAP-сообщений.

Ex: примеры вызовов операций RESTful WS

(DELETE): http://ex.org/deleteItem?id=1
(PUT): http://ex.org/createItem?id=2&name=”Item2”

Коды состояния HTTP-ответа RESTful WS может использовать для описания успешных ответов и исключений (аналог <Fault> для SOAP WS). Они описаны в Enum javax.ws.rs.core.Response.Status.

Пример JAR-RS WS

При проектировании RESTful WS надо начать с определения URI ресурсов. Для каждого класса реализации ресурсов JAX-RS WS рекомендуется создать интерфейс, который и будет нести всю метаинформацию. Применение аннотаций к интерфейсу позволит избавить реализацию от лишнего, делает возможным легкое изменение связанного с ней WS. А при отсутствии исходников, такая архитектура может быть вообще единственным способом экспорта POJO как компонента веб-службы.

В качестве примера рассматривается организация аукциона. Ресурсами являются ставки. Определение URI для операций CRUD:

- Операция GET. Получение списка ставок, полученных для данного клиента в указанной категории в период с даты начала и до даты конца:

```
/bidservice/list/{userID}/{category}?startDate=x&endDate=y
```

В данном URI часть параметров входит в путь (path), а часть — в параметрах URI HTTP-запроса.

- Операция POST. Добавить новую вставку, входные данные в формате XML предполагается отправлять в теле сообщения

```
/bidservice/addBid
```

- Операция GET. Вернуть ставку в формате XML или JSON

```
/bidservice/getBid/{bidID}
```

- Операция DELETE. Отменить вставку по указанному ID

```
/bidservice/cancel/{bidID}
```

В качестве реализации ресурса WS выбран SLSB, поскольку финансовые операции обычно требуют транзакций и безопасности. Для аннотирования выбирается @Local интерфейс SLSB. Класс SLSB: BidService. Интерфейс @Local: BidServiceRS

```
Ex:  интерфейс JAX-RS WS компонента на базе EJB

@Local @Path(“/bidService”)
public interface BidServiceRS {

  @POST
  @Path(“/addBid”)
  @Consumes(“application/xml”)
  public void addBid(Bid bid);

  @GET
  @Path(“/getBid/{bidID}”)
  @Produces({“application/json”,”application/xml”})
  public Bid getBid(@PathParam(“bidID”) long bidID);

  @DELETE
  @Path(“/delete/{bidID}”)
```

```
  public void cancelBid(@PathParam(“bidID”) long bidID);

  @GET
  @Path(“/list/{userID}/{category}”)
  public String listBids(
     @PathParam(“category”) String category,
     @PathParam(“userID”) long userID,
     @PathParam(“startDate”) String startDate,
     @PathParam(“endDate”) String endDate);
}
```

В спроектированном интерфейсе в методах addBid и getBid предполагается обмен сложным объектом Bid. Причем для addBid клиент должен передать его XML-представление, а для getBid веб-служба должна сформировать либо его XML-представление, либо его JSON-представление, в зависимости от запроса клиента.

Rem: В реальной жизни Bid может быть сложным объектом, встроенным в иерархию, в таком случае желательно создать на его базе простую версию — DTO.

Для автоматического связывания Bid с XML нужно разметить его аннотациями JAXB либо при помощи XSD, либо вручную. Поскольку в данном примере Bid – простой DTO, то достаточно нескольких аннотаций JAXB:

```
Ex:  JAXB класс (бин) Bid
@XmlRootElement(name=“Bid”)
@XmlAccessorType(XmlAccessType.FIELD)
public class Bid {
  @XmlElement
  private XMLGrigorianCalendar bidDate;
  @XmlAttribute
  private Long bidID;
  @XmlElement
  private double bidPrice;
  public Bid(){};
}
```

```
Ex: пример XML-представления объекта Bid
<Bid bidID=”10”>
  <bidDate>2012-05-18T20:01:33.088-03:00</bidDate>
  <bidPrice>11.0</bidPrice>
</Bid>
```

JSON-представление:

```
Ex: пример JSON-представления объекта Bid
{
   “@bidID” : “10”,
   “bidDate” : “2012-05-18T20:01:33.088-03:00”,
   “bidPrice” : “11.0”
```

```
}
```

Rem: хотя JSON, завоевавший большую популярность в RESTful WS, внесен в стандартные MIME-типы JAX-RS, в JEE7 его связывание с Java-объектами все еще не стандартизовано. Поэтому для работы с JSON-представлениями нужно подключать вендор-специфические реализации JSON-обработки вроде EclipseLink MOXy (Glassfish 4).

Реализация SLSB ресурса в данном примере роли не играет, поэтому она опущена. Для объединения всех ресурсов и вспомогательных классов поставщиков в единую RESTful службу используется класс-конфигуратор, расширяющий абстрактный javax.ws.rs.core.Application, который можно аннотировать @ApplicationPath для задания общей части URI для всех ресурсов веб-службы.

Ex: пример класса-конфигуратора JAX-RS

```
@javax.ws.rs.ApplicationPath("resources")
public class ApplicationConfig extends javax.ws.rs.core.Application {
  @Override
  public Set<Class<?>> getClasses() {
    Set<Class<?>> classes = new HashSet();
    classes.add(BidServiceRS.class);
    return classes;
  }
}
```

Класс-конфигуратор Application, ресурсы, поставщики упаковываются в war-архив, класс-конфигуратор регистрируется в системном сервлете JAX-RS.

Протестировать полученную веб-службу можно при помощи простого браузера, введя соответствующий URI, полученный конкатенацией серверного пути, значения @ApplicationPath, значения @Path класса ресурса, значения @Path метода.

Ex: пример URI (NetBeans, Glassfish 4.1)
http://localhost:8080/<ApplicationName>/resources/bidService/getBid/15

В ответ браузер получит HTTP-сообщение с XML-представлением объекта Bid.

Аннотация @Path

Аннотация @javax.ws.rs.Path - это основная аннотация JAX-RS уровня типа и метода. На уровне типа она определяет класс или интерфейс как ресурс JAX-RS. Классом ресурса может быть любой класс POJO, удовлетворяющий следующим требованиям :

- класс должен быть аннотирован @Path
- класс должен быть публичным и не абстрактным
- должен присутствовать публичный конструктор без аргументов (по-умолчанию)

- не должен быть перегружен метод finalize
- объекты класса не должны сохранять клиентское состояние между вызовами

В качестве ресурсов JAX-RS могут использоваться и EJB: @Singleton и SLSB.

На уровне метода @Path добавляет дополнительный путь для соответствующей операции. Единственный строковый параметр value определяет относительный путь к операции или ресурсу WS. При наличии @Path на уровне класса и метода их значения value конкатенируются. В значении пути value можно указать параметры, связанные с параметрами метода и автоматически извлекаемые при запросах. Для этого в строке value можно определить конструкции вида {xxx} или {xxx : yyy}, где xxx – алиас параметра, а yyy – регулярное выражение, по которому будут отфильтровываться допустимые параметры. Связь параметра метода с параметрами пути из значения value устанавливается при помощи @PathParam("xxx") уровня параметра метода.

```
@Local @Path(“/bidService”)
public interface BidServiceRS {
…
@DELETE
@Path(“/delete/{bidID}”)
public void cancelBid(...);

// Пример URI: http://xxx.org/ExJAXRSWS/resources/bidService/delete/777
```

Если метод не аннотирован @Path, то используется основной путь. При наличии одинаковых путей, методы выбираются по типу операции.

Аннотация @GET

Аннотация @javax.ws.rs.GET - это аннотация-метка без параметров уровня метода, задающая операцию WS и метод ее реализации, связанные с HTTP-запросом GET, служащего для получения представления ресурса. Метод может возвращать объект, MIME-представление которого передается клиенту в теле HTTP-ответа. Параметры обычно передаются в пути и параметрах URI запроса. При успешном выполнении GET должен приписывать ответу код 200 — хорошо.

Аннотация @POST

Аннотация @javax.ws.rs.POST – это аннотация-метка без параметров для пометки метода, вызываемого в ответ на HTTP-запрос POST, служащий для создания нового ресурса на сервере. В этом запросе на сервер могут передаваться крупноразмерные текстовые или бинарные (картинки, видео и т.п.) данные, MIME-тип которых определяется @Consumes. В ответ может посылаться URI вновь созданного ресурса. При успешном выполнении ответу приписывается код 201 — создано с URI нового ресурса. Если сервер не хочет возвращать URI, ответу приписывается код 204 (?).

Аннотация @PUT

Аннотация @javax.ws.rs.PUT – это аннотация-метка без параметров для пометки метода, вызываемого в ответ на HTTP-запрос PUT, служащего для обновления на сервере ресурса по заданному URI. Метод похож на POST, но идемпотентен, т.е. старый ресурс перезаписывается новым значением. В ответ может отдаваться представление обновленного ресурса. При успешном выполнении обновления уже существующего ресурса ответу должен быть приписан один из следующих кодов состояния: 200 — хорошо или 204 — нет содержимого, если сервер не хочет ничего возвращать.

Аннотация @DELETE

Аннотация @javax.ws.rs.DELETE – это аннотация-метка для метода удаления ресурса, идентифицированного строкой URI HTTP-запроса DELETE. Если удаление успешно завершено, то ответу может быть приписан код: 200 — хорошо (если в ответе содержится объект), 202 — принято (если удаление еще не начато), 204 — нет содержимого (в ответе отсутствует объект).

Аннотации @HEAD и @OPTIONS

Аннотация @HEAD помечает метод операции HEAD. Если такового метода не определено, то будет вызываться @GET, но ответ пойдет без HTTP-тела. Аннотация @IPTIONS помечает метод операции OPTIONS. Если такого метода нет, то будут автоматом возвращены метаданные запроса.

Параметры запроса

Параметры метода ресурса связаны с входными HTTP-запросами при помощи ряда аннотаций @*Param, @DefaultValue, @Consumes. Один не аннотированный параметр может представлять тело HTTP-запроса POST или PUT. HTTP-заголовок запроса Content-Type должен соответствовать значению @Consumes, иначе метод не вызовется и будет выброшено исключение (?).

Аннотация @PathParam

Аннотация @javax.ws.rs.PathParam – аннотация уровня параметра, поля, метода, работающая в связке с @Path. Она связывает параметры в пути URI с параметрами метода посредством алиаса, задаваемого в ее параметре value. Реализация JAX-RS должна автоматически выполнять преобразование типов при вызове метода.

```
@Path(“/delete/{bidID}”)
public void cancelBid(@PathParam(“bidID”) long bidID);

// Пример URI: http://xxx.org/ExJAXRSWS/resources/bidService/delete/777
// bidID = 777
```

```
@GET
@Path(“/get/{login : [a-zA-Z]+[\\d]+}”)
public void cancelBid(@PathParam(“login”) String log);

// Пример URI: http://xxx.org/ExJAXRSWS/resources/bidService/login/user1234
// bidID = user1234
```

Аннотация @QueryParam

Аннотация @javax.ws.rs.QueryParam – аннотация уровня параметра, поля, метода, работающая в связке с @Path. Она извлекает параметр запроса URI (…?xxx&yyy) по заданному в value имени и инициализирует им аннотируемый параметр.

```
public void getItem(@QueryParam(“id”) String id);

// Пример URI: http://xxx.org/ExJAXRSWS/resources/exResource/getItem?id=item11&type=2
// id = item11
```

Rem: при аннотировании Java-параметра надо убедиться, что корректно работает его преобразование в String и обратно. В частности, есть проблемы с таким преобразованием у java.util.Date, поэтому вместо него приходиться использовать строку.

Rem: некоторые реализации вроде Apache CXF позволяют включать расширения для регистрации обработчиков, которые могут решить проблему конвертации.

Аннотация @MatrixParam

Аннотация @javax.ws.rs.MatrixParam — аналогична @QueryParam, но извлекает ”матричный параметр”, представляющий пары, разделенные символом “;”.

```
public void exMethod(@MatrixParam(“Id”) String id, @MatrixParam(“Type”) int type);

// Пример URI: http://xxx.org/ExJAXRSWS/resources/exResource/getItem;Id=item11;Type=2
// id = “item11”; type =  2
```

Аннотация @CookieParam

Аннотация @javax.ws.rs.CookieParam – аннотация, вытаскивающая параметр из заголовка Cookie HTTP-запроса. Может использоваться для эмуляции сессии, если это необходимо.

```
public void exMethod(@CookieParam(“sessionID”) String id);

// Пример HTTP-заголовка Cookie
…
Cookie: sessionID=2DF234; param1=value1; param2=value2
```

```
// id = “2DF234”
```

Аннотация @HeaderParam

Аннотация @javax.ws.rs.HeaderParam вытаскивает параметр из HTTP-заголовка запроса.

```
public void exMethod(@CookieParam(“User-Agent”) String userAgent);

// Пример HTTP-заголовка User-Agent
…
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
// userAgent = “Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)”
```

Аннотация @FormParam

Аннотация @javax.ws.rs.FormParam используется в методах @PUT, @POST для извлечения параметра из формы, находящейся в теле запроса.

```
public void exMethod(@FormParam(“Id”) String id);

// Пример формы в HTTP-теле
…
Content-Disposition: form-data; name="Id"

test22
…
// id = “test22”
```

Аннотация @DefaultValue

Аннотация @javax.ws.rs.DefaultValue(“xxx”) может применяться совместно с любой аннотацией @*Param и задает значение параметра по-умолчанию.

```
public Item getItem(@DefaultValue(“0”) @QueryParam(“price”) int price) {…}
```

Аннотации @Produces и @Consumes. Использование MIME

Каждая операция JAX-RS может принимать или возвращать значения в различных представлениях, называемых MIME-типами (медиатипами). Для связи Java-объектов и MIME представлений используются т.н. поставщики сущностей (entity provider). Классы этих поставщиков связываются с классами JAX-RS ресурсов при помощи аннотаций @Produces и @Consumes. Стандартные поставщики входят в реализацию

JAX-RS.

Аннотация @javax.ws.rs.Produces – аннотация уровня метода и типа. Она определяет типы MIME, которые веб-служба может производить и передавать клиенту. Атрибутом value является String[], где перечисляются MIME-типы, доступные клиенту. Если операция способна производить более одного MIME-типа представления, то конкретный тип определяется HTTP-заголовком Accept клиентского запроса.

Реализация JAX-RS должна автоматически преобразовывать Java-результат в требуемый MIME-тип. Основными из стандартных для JAX-RS MIME-типов являются:

- application/xml
- application/json
- text/plain
- application/octet-stream (произвольные двоичные данные)

Стандартные медиатипы перечислены в виде строковых констант в javax.ws.rs.core.MediaType.

```
@GET
@Produces({“application/json”,”application/xml”})
public Bid getBid(@PathParam(“bidID”) long bidID);
```

Rem: при выборе application/xml преобразование будет выполнять JAXB, поэтому класс Java-объекта должен быть правильно размечен.

Аннотация @javax.ws.rs.Consumes – аннотация уровня метода и типа. Она указывает MIME-типы, которые JAX-RS может принимать в качестве параметра и преобразовывать в Java-параметры метода. Атрибутом value является String[], где перечисляются MIME-типы, допустимые для тела (HTTP-объекта) запроса. Стандартные медиатипы перечислены в виде строковых констант в javax.ws.rs.core.MediaType.

```
@POST
@Consumes({MediaType.APPLICATION_XML})
public void addBid(Bid bid);
```

Rem: стандартно для @Consumes и @Produces уровень метода приоритетнее уровня типа, что удобно для задания на уровне класса или интерфейса MIME представления по-умолчанию и переопределения его в методах.

При отсутствии аннотаций @Consumes и @Produces считается, что соответствующий MIME-тип может быть произвольного формата */*. На стороне JAX-RS WS это требует реализации потребительских поставщиков всех стандартных MIME-типов. На стороне клиентских браузеров заголовок ”Content-Type: */*” обычно приводит к скачиванию HTTP-объекта в файл.

Аннотация @Provider. Поставщики (entity poviders)

Поставщик сущности (entity provider) – компонент, занимающийся преобразованием Java-объекта в какое-либо представление и наоборот. В качестве примера можно привести JAXB, связывающий POJO с XML-документом.

Для незаметных преобразований между представлениями основных MIME-типов и объектами ряда Java-типов существуют стандартные поставщики, входящие в состав JAX-RS. Например, XML связан с JAXB-компонентами (оболочечными JAXBElement или аннотированными JAXB-аннотациями POJO), также в JEE7 определено объектное связывание с JSON для JAX-RS, реализуемое вендорами.

Допустимые входные и выходные MIME-типы представлений, преобразуемые в Java-параметр и получаемые из Java-ответа соответствующего метода JAX-RS ресурса, определяются при помощи @Consumes и @Produces, соответственно. Если есть стандартный поставщик, то преобразование с его помощью происходит неявно. Но если используется нестандартный формат или объект, то необходимо определить свои поставщики: потребительский и производящий.

Потребительский (@Consumes) поставщик преобразует входное представление одного из заданных MIME-типов в Java-объект указанного класса. Он представляет собою класс, удовлетворяющий следующим требованиям:

- реализует интерфейс javax.ws.rs.ext.MessageBodyReader<T>, где T – Java-класс, связываемый с медиатипами
- аннотирован @javax.ws.rs.ext.Provider
- необязательно может аннотироваться @Consumes({“aaa/bbb, …, “yyy/zzz”}), где aaa/bbb … yyy/zzz – медиатипы для связи с классом T. По-умолчанию предполагается возможность связывания любого медиатипа.

Производящий (@Produces) поставщик преобразует выходной Java-объект заданного класса в представление одного из указанных MIME-типов. Он представляет собою класс, удовлетворяющий следующим требованиям:

- реализует интерфейс javax.ws.rs.ext.MessageBodyWriter<T>, где T – Java-класс, связываемый с медиатипами
- аннотирован @javax.ws.rs.ext.Provider
- необязательно может аннотироваться @Produces({“aaa/bbb, …, “yyy/zzz”}), где aaa/bbb … yyy/zzz – медиатипы для связи с классом T. По-умолчанию предполагается возможность связывания любого медиатипа.

Методы @Provider класса могут выбрасывать непроверяемое исключение javax.ws.rs.WebApplicationException при невозможности создания представления.

Ниже пример определения и использования самопального формата ресурса Item.

Ex: самопальное текстовое и Java-представления ресурса Item

Самопальное текстовое представление Item - id/Name/size:
11/test/4

Java-представление Item — класс POJO Item

```
public class Item {
  private String id;
  private String name;
  private int size;
  public String getId() {return id;}
  public void setId(String id) {this.id=id;}
  public String getName() {return name;}
  public void setName(String name) {this.name=name;}
  public int getSize() {return size;}
  public void setSize(int size) {this.size=size;}
}
```

Ex: пример производящего поставщика

```
// Параметризованный класс поставщика. Параметром класса выступает класс POJO Item
@Provider
@Produces("custom/format")
public class ExItemWriter implements MessageBodyWriter<Item> {
  String charset = “UTF-8”;
// проверка, является ли преобразуемый объект типом Item для связывания
  @Override
  public boolean isWriteable(Class<?> type, Type genericType, Annotation[] annotations, MediaType
mediaType) {
    return Item.class.isAssignableFrom(type);
  }
// создание текстового представления объекта Item в формате: ”id/name/size”
  @Override
  public void writeTo(Item item, Class<?> type, Type gType,  Annotation[] a, MediaType mt,
MultivaluedMap<String, Object> httpHeaders, OutputStream outputStream) throws IOException,
WebApplicationException {
    outputStream.write(item.getId().getBytes(charset));
    outputStream.write(“/”.getBytes(charset));
    outputStream.write(item.getName().getBytes(charset));
    outputStream.write(“/”.getBytes(charset));
    outputStream.write(String.valueOf(item.getSize()).getBytes(charset));
  }
// Получение размера представления ответа в байтах
  @Override
  public long getSize(Item item, Class<?> type, Type gType, Annotation[] a, MediaType mt) {
    int dS = “/”.getBytes(charset).length;
    return item.getId().getBytes(charset).length + dS + item.getName().getBytes(charset).length + dS +
String.valueOf (item.getSize()).getBytes(charset).length;
```

```
  }
}
```

Ex: пример потребительского поставщика

```
// Параметризованный класс поставщика. Параметром класса выступает класс POJO Item
@Provider
@Consumes("custom/format")
public class ExItemReader implements MessageBodyReader<Item> {
// проверка, приводится ли преобразуемый объект к типу Item для связывания
   @Override
  public boolean isReadable(Class<?> type, Type gType, Annotation[] a, MediaType mt) {
    return Item.class.isAssignableFrom(type);
  }
// проверка, является ли преобразуемый объект типом Item для связывания
  @Override
  public Customer readFrom(Class<Item> type, Type gType, Annotation[] a, MediaType mt,
      MultivaluedMap<String, String> httpHeaders, InputStream inputStream)
      throws IOException, WebApplicationException {
    String[] str = stream2String(inputStream).split("\/");
    Item item = new Item();
    item.setId(str[0]);
    item.setName(str[1]);
    item.setSize(Integer.valueOf(str[2]));
    return Item;
  }
  private String stream2String(InputStream inputStream){
    ByteArrayOutputStream result = new ByteArrayOutputStream();
    byte[] buffer = new byte[1024];
    int length;
    while ((length = inputStream.read(buffer)) != -1) {
      result.write(buffer, 0, length);
    }
    return result.toString("UTF-8");
  }
}
```

Ex: фрагмент класса ресурса JAX-RS

```
…
@Context
private UriInfo uriInfo;

// Метод getItem привязан к поставщику ExItemWriter посредством "custom/format"
// Строковый параметр id заполняется стандартным поставщиком из параметра пути URI - ID
@GET
@Produces({"custom/format"})
@Path("{ID}")
public Item getItem(@PathParam("ID") String id){
  Item item = …;
  return item;
}
```

```
// Метод createItem привязан к поставщику ExItemReader посредством “custom/format”
// Ответный объект URI преобразуется в текст “text/plain” стандартным поставщиком
@POST
@Consumes({“custom/format”})
@Produces({“text/plain”})
public URI  createItem(Item item){
  …
  URI itemURI = uriInfo.getAbsolutePathBuilder().getPath(item.getId()).build();
  return itemURI;
}
…
```

Часть таблицы связывания стандартных поставщиков:

Java-тип	**Описание и MIME-типы**
byte[]	*/*
java.lang.String	*/*
java.io.InputStream	*/*
java.io.Reader	*/*
java.io.File	*/*
javax.ws.rs.core.StreamingOutput	*/* (только MessageBodyWriter)
javax.xml.bind.JAXBElement (@RootElement)	text/xml, application/xml

Возвращение ответа. Класс Response

Метод, реализующий операцию JAX-RS может возвращать void, Response, GenericEntity и любой Java-тип, обладающий подходящим представлением, задающимся аннотациями @Produces. Это представление передается в теле HTTP-сообщения (HTTP-объекте).

HTTP-заголовок Accept должен соответствовать значению @Produces, иначем будет выброшено исключение. Реализация JAX-RS определяет MIME-тип HTTP-объекта и при помощи соответствующего поставщика (Provider) создает из этого результирующего объекта нужное представление. Пустое значение void и все типы с пустым представлением возвращаются по-умолчанию с кодом 204, при непустом теле — с кодом 200. В Response код можно переопределять.

Если требуется добавить в ответ некую дополнительную метаинформацию - код ответа, HTTP-заголовки, параметры Cookie, то поставщику надо отдать результирующий объект в классе-обертке javax.ws.rs.core.Response.

Rem: вообще рекомендуется всегда возвращать класс-оболочку Response, поскольку это дает возможность более гибкой настройки ответа.

Для создание объектов Response удобно использовать шаблон ”текущий интерфейс” при помощи вложенного в него статического класса Response.ResponseBuilder.

```
@GET
@Produce({MediaType.APPLICATION_JSON})
public Response getItem(String id, String name, int size){
  return Response.ok(new Item(id,name,size), MediaType.APPLICATION_JSON).build();
}
```

Малая часть методов «текучего интерфейса» Response:

- created(URI) - создает объект ResponseBuilder для созданного ресурса (с его URI)
- noContent() - создает объект ResponseBuilder для пустого ответа
- ok(...) - создают объект ResponseBuilder с состоянием 200 — Хорошо
- serverError() - создает объект ResponseBuilder с состоянием 500 (ошибка сервера)
- status(int) - создает объект ResponseBuilder с предоставленным состоянием

Некоторые методы работы с метаинформацией ответа Response:

- getCookies() - дает cookie из сообщения ответа
- getHeaders() - дает заголовки из сообщения ответа
- getStatus() - дает код состояния, ассоциированный с ответом
- readEntity(...) - методы, дающие сообщение ответа в виде Java-объекта, получаемого при помощи поставщика MessageBodyReader

Некоторые методы фабрики Response.ResponseBuilder

- build() - создает ответ Response
- cookie(NewCookie) – добавление новой куки к ответу
- header(String,Object) – добавляет заголовок ответа
- status(int) - задает код состояния ответа
- entity(...) - методы, задающие Java-объектом сообщение ответа, MIME-представление которого формируется поставщиком MessageBodyWriter

Ex: примеры построения ответов Response “текучим интерфейсом”

```
Response.ok().build();
Response.ok().cookie(new NewCookie("SessionID", "14AAF3")).build();
Response.ok("Plain Text").expires(new Date()).build();
Response.ok(new Item(“id_22”,”Simple Item”,2), MediaType.APPLICATION_JSON).build()
Response.noContent().build();
```

Построение URI. Класс UriBuilder

Для обеспечения связанности приложения, ресурсы RESTful WS должны предоставлять клиенту URI для перехода. С этой целью можно применять javax.ws.rs.core.UriBuilder,

помогающий строить объекты java.net.URI при помощи «текучего интерфейса».

Ex: примеры построения ответов URI “текучим интерфейсом”

```
java.net.URI uri;
// Ex. uri: http://www.ex.org/item/777
uri = UriBuilder.fromUri("http://www.ex.org").path("item").path("777").build();
// Ex. uri:  http://www.ex.org/item?name=item_1
uri = UriBuilder.fromUri("http://www.ex.org").path("item ") .queryParam("name", "item_1").build();
// Ex. uri:  http://www.ex.org/item;name=item_1
uri = UriBuilder.fromUri("http://www.ex.org").path("item") .matrixParam("name", "item_1").build();
// Ex. uri:  http://www.ex.org/item?name=item_1
uri = UriBuilder.fromUri("http://www.ex.org").path("{path} ").queryParam("name",
"{value}").build("item", "item_1");
// Ex. uri:  /bidService/777
uri = UriBuilder.fromResource(BidService.class).path("777").build();
// Ex. uri:  http://www.ex.org/#777
uri = UriBuilder.fromUri("http://www.ex.org").fragment("777").build ();
```

Аннотация @Context. Контекст запроса

Метаинформацию ответа можно формировать при помощи объекта Response. Метаинформация запроса сохраняется реализацией JAX-RS в ряде объектов классов javax.ws.rs.core:

- HttpHeaders – HTTP-заголовки запроса, включая Cookie
- UriInfo – информация об URI запроса
- Request
- SecurityContext – контекст безопасностии, в т.ч. getUserPrincipal, isUserInRole

Объекты этих классов можно внедрять в свойства и параметры методов специальной аннотацией @javax.ws.rs.core.Context.

Ex: примеры использования @Context для передачи клиенту HTTP-заголовков его запроса и абсолютного пути к созданному ресурсу

```
@Path("/metainfo")
public class ExJAXRSWS {
  @Context
  UriInfo uriInfo;
  @Inject
  private ItemEJB itemEJB;
  @GET
  @Path("media")
  public String getDefaultMediaType(@Context HttpHeaders headers) {
    List<MediaType> mediaTypes = headers.getAcceptableMediaTypes();
    return mediaTypes.get(0).toString();
  }
  @GET
```

```
 @Path("language")
 public String getDefaultLanguage(@Context HttpHeaders headers) {
  List<String> acceptLanguages = headers.getRequestHeader(HttpHeaders.ACCEPT_LANGUAGE);
  return acceptLanguages.get(0);
 }
 @POST
 @Consumes(MediaType.APPLICATION_XML)
 public Response createItem(Item item) {
  Item attachedItem = itemEJB.persist(item);
  URI itemUri = uriInfo.getAbsolutePathBuilder().path(attachedItem .getId()).build();
  return Response.created(itemUri).build();
 }
}
```

Rem: в JAX-RS 2.0 объекты перечисленных классов пока не интегрированы с CDI, т.е. их нельзя внедрить напрямую при помощи @Inject и приходится использовать специальную аннотацию @Context. Можно использовать костыль в виде CDI-управляемого класса-производителя CDI, позволяющий использовать @Inject для неинтегрированных с CDI объектов аналогично внедрению клиентской SEI-заглушки JAX-WS при помощи @WebServiceRef.

Исключения и поставщик ExceptionMapper

Как и любая другая часть приложения, компоненты JAX-RS могут давать сбои: из-за невалидных запросов и сетевых проблем, ошибок в бизнес-логике.

Исключения JAX-RS компонентов, допускаемые контейнером — непроверяемые (RuntimeException). Все проверяемые исключения перед выбросом должны быть перепакованы в одно из допустимых. В JAX-RS для этого определено непроверяемое исключение javax.ws.rs.WebApplicationException и куча его уточняющих наследников. Также можно использовать и более общие исключения, например, javax.ws.WebServiceException (непроверяемое, из JSE) или javax.servlet.ServletException (проверяемое, из JEE).

Исключения WebApplicationException и его наследники будут автоматически обрабатываться реализацией JAX-RS и формировать при помощи внутреннего поля Response HTTP-ответ с соответствующим кодом статуса (аналог <Fault> в SOAP WS). Объект WebApplicationException при создании можно инициализировать кодами статуса ответа, определенными в javax.ws.rs.core.Response.Status.

```
Ex:  пример использования исключений

@Path("/items")
public class ExJAXRSWS {
 @Inject
 private ExEJB exEJB;
 @Path("{itemId}")
 public Item getItem(@PathParam("itemId") Long id) {
```

```
// Проброс стандартного непроверяемого исключения
    if (id < 0)  throw new IllegalArgumentException("Id < 0!");
    Item item = exEJB.find(id);
// Проброс специального исключения WebApplicationException с кодом 404
    if (item == null) throw new WebApplicationException(Response.Status.NOT_FOUND);
    return customer;
  }
}
```

Для удобства можно определить поставщика исключений, который будет перехватывать исключительные ситуации и обрабатывать их, формируя свой ответ Response, нужное представление которого будет возвращаться клиенту. Поставщик исключения должен реализовывать параметризованный javax.ws.rs.ext.ExceptionMapper<E extends java.lang.Throwable> и снабжаться аннотацией @Provider. Реализация JAX-RS при выбросе исключения, используя его тип, найдет по параметру E соответствующий поставщик и вызовет его.

```
Ex:  пример поставщика, обрабатывающего исключение JPA. Класс поставщика параметризован
EntityNotFoundException

@Provider
public class EntityNotFoundMapper implements ExceptionMapper<EntityNotFoundException> {
  @Override
  public Response toResponse(javax.persistence.EntityNotFoundException e) {
    return Response.status(404).entity(e.getMessage()).type(MediaType.TEXT_PLAIN).build();
  }
}
```

Жизненный цикл JAX-RS компонента

Компоненты JAX-RS не сохраняют состояния и строятся при получении запроса, что гарантирует бесконфликтность при множественных параллельных запросах в WS. При развертке на JEE-сервере как и любым другим компонентам, компонентам JAX-RS доступны аннотации жизненного цикла @PreDestroy и @PostConstruct, описанные в JSR 250 (Common Annotations for the Java Platform). Также доступны перехватчики CDI или EJB, в зависимости от типа компонента, также влияющие на жизненный цикл в точках входа и выхода из методов.

Упаковка JAX-RS WS

Веб-служба JAX-RS WS организуется как сервлетное приложение. Упаковываются класс-конфигуратор Application, ресурсы (@Path типы), нестандартные поставщики (@Provider классы) и их вспомогательные классы. Класс-конфигуратор должен быть зарегистрирован в системном сервлете JAX-RS. Само приложение упаковывается в war-архив. В зависимости от реализации все это либо собирается автоматически из аннотаций, либо вручную, с созданием дескриптора WEB-INF/web.xml. Для Jersey (стандартная реализация JAX-RS в Glassfish) XML-фрагмент веб-службы в web.xml

будет иметь примерно такой вид:

```
Ex: XML-фрагмент RESTful веб-службы в web.xml для реализации Jersey

<servlet>
  <description>JAX-RS Tools Generated — Do not modify</description>
  <servlet-name>JAX_RS_Example</servlet-name>
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>....ApplicationConfig
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>JAX_RS_Example</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Системных артефактов, в отличие от JAX-WS, не присутствует. Кроме этого спецификацией никак не определяется XML дескриптор развертки.

Клиентский API

Клиентский API появился с JAX-RS 2.0. Он позволяет строить запросы при помощи текучего интерфейса, а также обрабатывать ответы. Клиент может быть построен на базе любого JEE-приложения и работать в EJB-контейнере или в Web-контейнере (CDI-компоненты). Либо на базе JSE-приложения, при условии, что в него будет включена реализация JAX-RS. API находится в пакете javax.ws.rs.client.

Rem: вообще запросы вроде POST или GET можно делать из обычного браузера, для остальных запросов можно поставить специальные плагины или использовать утилиты вроде консольного cUrl.

Запросы в текучем интерфейсе можно делать множеством способов. Пример построения запроса и получения ответа:

```
Ex:  пример простого GET-запроса к гипотетическому REST WS по адресу ex.org/item,
возвращающему ресурс в обычном текстовом представлении.

// Традиционная запись
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target("http://ex.org/item");
Invocation.Builder invocationBuilder = webTarget.request(MediaType.TEXT_PLAIN);
Invocation invocation = invocationBuilder.buildGet();
Response response = invocation.invoke();
```

```
// Текучий интерфейс
Response response =
ClientBuilder.newClient().target(“http://ex.org/item”).request(“text/plain”).buildGet().invoke();
```

Текучий клиентский API состоит из следующих классов и интерфейсов:

Тип	**Описание**
ClientBuilder	Класс фабрики клиентов фабрики клиентов
Client	Входная точка цепочки запрос-ответ. Задает общий URI запроса и создает WebTarget методом target(...). Настраивает свой контекст расширяя Configurable<Client>.
WebTarget	Целевая точка. Уточняет URI запроса, формируя путь, параметры. Создает Invocation.Builder методом request(...). Настраивает свой контекст расширяя Configurable<WebTarget>
Invocation.Builder	Построитель запроса. Задает метод, MIME-тип, Cookie и другие заголовки. Создает Invocation методом build(...).
Invocation	Запрос. Вызывает синхронно или асинхронно REST WS. Синхронный запрос invoke(...) создает ответ Response.
Configurable<C>	Конфигуратор контекста параметрического типа C. Регистрирует классы или объекты поставщиков и других вспомогательных классов, задает свойства. Меняет контекст через Configuration (терминально).
Entity	Класс-оболочка для объектов сообщения (методы POST и PUT)

Создание клиента client с регистрацией класса поставщика потребителя CustomReader.class и свойства контекста вызова :

```
Client client = ClientBuilder.newClient().register(CustomReader.class).property("exProp", 777);
```

Создание целевой точки target по URI http://ex.org/item с его уточнением до http://ex.org/item/xxx?par=11 :

```
WebTarget target = client.target(new URI(“http://ex.org/item”)).path("xxx").queryParam("par",11);
```

Создание GET-запроса invocationGet с заголовками Accept: application/xml и Accept-Language: ru :

```
Invocation invocationGet = target.request("application/json").acceptLanguage("ru").buildGet();
```

Создание POST-запроса invocationPost с передачей в теле XML-представления объекта Item :

```
Invocation invocationPost = target.request().buildPost(Entity.entity(new Item(),"application/xml"));
```

Запросы GET и POST и получение ответов:

```
Response responsePost =  invocationPost.invoke();
Item item = invocationGet.invoke().readEntity(Item.class);
```

Текучий интерфейс (используется укороченный вариант синхронного вызова .get(Item.class) вместо .buildGet().invoke().readEntity(Item.class)):

```
Response responseGet =  ClientBuilder.newClient().register(CustomReader.class).property("exProp",
777).target(new URI(“http://ex.org/item”)).path("xxx").queryParam("par",11).
request("application/json").acceptLanguage("ru").get(Item.class);
```

Для обработки полученных ответов используется уже упомянутый на серверной стороне javax.ws.rs.core.Response. Клиент использует его как контейнер, из которого можно получить тело сообщения (HTTP-объект) и заголовочную информацию, в т.ч. код ответа, длину ответа, Cookie:

```
response.getStatusInfo() // == Response.Status.OK
response.getLength() // == 4
response.getDate()  // ...
response.getHeaderString("Content-type") // .equals("text/plain"));
```

Тело сообщения может быть получено методами response.getEntity(…). Этот метод конвертирует входной поток данных в заданный Java-объект. Для конвертации используется один из поставщиков MessagBodyReader – стандартных или зарегистрированных при формировании объекта Client. Выбор поставщика производится автоматически, по типу целевого Java-объекта.

Автоматическая конвертация HTTP-объекта с текстовым представлением ответа в String и HTTP-объекта с XML-представлением в JAXB-компонент Item стандартными поставщиками:

```
String body = response.readEntity(String.class);
Item item = response.readEntity(Item.class);
```

Эффективное использование JAX-RS в EJB

RESTful WS близки к сетевому протоколу HTTP, что выражается в необходимости адаптации Java-интерфейса к таким форматам, как JSON или XML. Наиболее беспроблемными являются интерфейсы с методами, принимающими и отдающими простые типы и String.

Метаданные JAX-RS целесообразно выносить в Java-интерфейсы, т.к. это дает гибкость в модификации WS без затрагивания реализации (а если нет исходников реализации, то это может быть единственным способом экспорта в JAX-RS веб-службу). Кроме того, технические аннотации не загрязняют бизнес-логику.

Для экспорта EJB удобно использовать шаблон адаптер с отдельным интерфейсом. Адаптер можно сделать из CDI-компонентов (POJO) с внедрением в него EJB. При этом всю специфику и ограничения RESTful адаптер возьмет на себя: использование облегченных DTO, преобразования типов и прочее. При этом EJB останется неизменным.

При реализации RESTful веб-службы следует придерживаться ее семантики: метод извлечения данных должен вызываться операцией GET, метод удаления — операцией DELETE. Должны поддерживаться безопасности и идемпотентность реализуемых методов.

RESTful WS должны быть простыми, одним из признаков простоты является возможность их запроса из браузера. Если же для общения с сервером нужен специализированный клиент, то это повод для перехода на другие удаленные технологии вроде SOAP WS или RMI.

SOAP vs REST

SOAP является полновесной объектной распределенной системой. SOAP – самодокументируемая технология, использование стандартного WSDL позволяет полностью автоматизировать процесс создания клиента и самой веб-службы. Однако тут могут быть подводные камни с межоперабельностью. SOAP WS благодаря объектной ориентированности модели может поддерживать окружение безопасности и транзакционный механизм. SOAP допускает маршрутизацию через промежуточные узлы, каждый из которых может исследовать заголовки SOAP-сообщения, выполнять нужные операции и пересылать сообщение далее. В Java технология SOAP WS полностью стандартизована и внесена в JSE как JAX-WS.

RESTful WS является более простой формой веб-служб, своеобразное возвращение к истокам (HTTP, Internet, www). RESTful использует HTTP как прикладной протокол и должна отображаться на его операции CRUD, что накладывает достаточно строгие ограничения на интерфейсы реализации. Благодаря параметрическим запросам, клиент может получать результат в разных форматах (представлениях). Применение компактных форматов JSON или BLOB уменьшает накладные расходы на пропускную способность и вычислительную сложность обработки, тем более ряд современных ОС оптимизирует работу с ними. В результате RESTful WS более масштабируемы. Хотя REST является всего лишь архитектурой проектирования, а не стандартом, простота RESTful служб позволила автоматизировать процесс их создания, а также клиентов к ним на платформах с развитым API. В Java технология RESTful WS входит в JEE, начиная с JEE7 JAX-RS позволяет создавать как сами WS, так и клиентов к ним.

Rem: для описания RESTful WS можно использовать язык WADL, появилась такая возможность и в WSDL (?)1.2(?), но эти возможности пока широко не используются.

JAXB

JAXB = Java Architecture for XML Binding (Java-архитектура для связывания с XML). Это фрэймворк для связи между XML-разметкой и Java-объектами. Связывание осуществляется при помощи аннотаций, помечающих соответствующий Java-класс.

Rem: наряду с аннотациями можно использовать и классы-обертки JAXB. Так, аналогом @javax.xml.bind.annotation.XMLRootElement является javax.xml.bind.JAXBElement<T>.

Эти аннотации расставляются либо вручную, либо генерируются (вместе с классом) по файлу схемы (XSD).

Ex : пример схемы XML (XSD) для ее связывания с Java-классом

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema" targetName
space="http://titan.com/custom">
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element name="street" type="xs:string" minOccurs="0"/>
      <xs:element name="city" type="xs:string" minOccurs="0"/>
      <xs:element name="zip" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

В результате разупорядочения (unmarshalling) XSD, JAXB создается класс Address:

Ex : результат разупорядочения XSD – класс Address (компилятор XJC)

```
// Связывание с XML всех не static, не transient полей
@XmlAccessorType(XmlAccessType.FIELD)
// Имя типа, порядок элементов
@XmlType(name = "address", propOrder = {"street","city","zip"})
public class Address {
    protected String street;
    protected String city;
    protected String zip;
    public String getStreet() {
        return street;
    }
    public void setStreet(String value) {
        this.street = value;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String value) {
        this.city = value;
    }
```

```
    public String getZip() {
        return zip;
    }
    public void setZip(String value) {
        this.zip = value;
    }
}
```

Простые объекты можно размечать аннотациями JAXB самостоятельно. Поскольку прикладной движок, занимающийся упорядочиванием (marshalling), синтезируется из метаинформации (аннотаций), никакой дополнительной XSD не нужно. Также этот прикладной движок годится для многократного использования, в том числе для обработки XML-запросов, что делает его подходящим для WS.

```
Ex:  JAXB класс (бин), размеченный для связи с XML-документом
// Корневой элемент
@XMLRootElement(name="Bid")
// Связывание с XML всех не static, не transient полей
@XmlAccessorType(XmlAccessType.FIELD)
// Для автоматического внесения классов-потомков в JAXB-контекст
@XmlSeeAlso(BidChild.class)
public class Bid {
//Связывание с отдельным XML-элементом
  @XmlElement
//Кроссплатформенный вариант Date, совместимый с .NET
  private XMLGrigorianCalendar bidDate;
//Связывание с атрибутом корневого элемента
  @XmlAttribute
  private Long bidID;
  @XmlElement
  private double bidPrice;
//Публичный конструктор по-умолчанию
  public Bid(){};
}
```

```
Ex:  пример XML-документа, связанного с объектом класса Bid
<Bid bidID="10">
  <bidDate>2012-05-18T20:01:33.088-03:00</bidDate>
  <bidPrice>11.0</bidPrice>
</Bid>
```

Типичный пример обертывания List<T> для получения XML-представления списка T:

```
Ex:  JAXB-компонент — обертка типизированного листа, для получения XML-представления
списка элементов Bid.

@XmlRootElement(name="Bids")
public class Bids extends ArrayList<Bid> implements List<Bid> {
  public Bids() {
```

```
    super();
  }
  public Bids(Collection<? extends Bid> c) {
    super(c);
  }
// Аннотация @XmlElement метода забьет корневые аннотации возвращаемых Bid
  @XmlElement(name = "Bid")
  public List<Bid> getBids() {
    return this;
  }
  public void setBids(List<Bid> bids) {
    this.addAll(bids);
  }
}
```

Пример XML-фрагмента:

```
<Bids>
  <Bid bidID="10">
    <bidDate>2016-07-20T15:23:20.467+03:00</bidDate>
    <bidPrice>9.0</bidPrice>
  </Bid>
  <Bid bidID="11">
    <bidDate>2016-07-20T15:34:11.32+03:00</bidDate>
    <bidPrice>14.5</bidPrice>
  </Bid>
</Bids>
```

Фрэймворк JAXB обладает значительным потенциалом для работы с почти любыми XML-схемами без самопального кода. Есть тем не менее некоторые ограничения, вызванные тем, что концепция XML-схемы создавалась без учета требований к строгому ее отображению на языки типа Java (XS-элементы union, restriction и т.п.). В таких случаях следует использовать специальные аннотации. Более детальную информацию по разрешению подобных проблем в JAXB можно получить из спецификации JSR-222, java-doc и специальной литературы.

JAXP

JAXP (Java API for XML Processing) - программный прикладной интерфейс для обработки XML). Это программный интерфейс для взаимодействия с анализаторами XML-документов, основанными на технологиях SAX, DOM, StAX. Он позволяет проводить полуавтоматическую обработку XML-документов и создавать их. Кроме того, JAXP поддерживает язык XSLT (eXtensible Stylesheet Language Transformations, расширяемый язык преобразования таблиц стилей), используемый для преобразования XML-документов.

Технология SAX (Simple API for XML) основана на последовательном чтении SAX-парсером XML-документа и создании при этом событий, связанных с возникновением

атрибутов и элементов. В ответ на каждое требуемое событие SAX-парсер вызывает соответствующий зарегистрированный метод-обработчик (handler) использующего его приложения. Регистрация идет по принципу обратного вызова, путем передачи SAX-парсеру ссылки на объект обработчиков перед началом работы. SAX-анализ потребляет немного ресурсов, но может использоваться лишь для операций чтения XML-документа. API располагается в пакетах javax.xml.parsers.* и org.xml.sax.*

В нижеприведенном примере SAX-обработки XML-документа создаются класс-обработчик SAXHandler и объект SAX-парсера, в метод парсинга parser.parse которого передаются входной поток XML-документа и объект-обработчик. Класс SAXHandler расширяет стандартный обработчик, который создает строковый буфер в ответ на начало документа, заполняет буфер содержимым отфильтрованных по имени элементов, выводит на печать буфер в ответ на конец документа.

Ex : пример SAX-анализа XML-документа

```
// Код SAX-парсинга
public void saxParsing(InputStream in) throws Exception {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();
    parser.parse(in, new SAXHandler());
}

// Класс обработчиков
public class SAXHandler extends org.xml.sax.helpers.DefaultHandler {
  StringBuffer strbf;
  String qName;
// Обработчик события  - начала XML-документа
  @Override
  public void startDocument () {
    strbf = new StringBuffer();
  }
// Обработчик события  - конца XML-документа
  @Override
  public void endDocument () {
    System.out.println(strbf);
  }
// Обработчик события — начала элемента
  @Override
  public void startElement(String uri, String localName, String qName, Attributes attributes) {
    this.qName=qName;
  }
// Обработчик события — содержимое элемента
  @Override
  public void characters(char[] ch, int start, int length) {
    if(qName.equals("name") || qName.equals("phone") || qName.equals("address")) {
      strbf.append(ch,start,length);
    }
  }
}
```

Технология DOM (Document Object Model) основана на построении в памяти универсального объектного дерева по XML-документу. Дерево состоит из набора объектов стандартных классов (узлов, атрибутов, текстовых элементов и т.д.). DOM-анализ дает полное представление XML-документа, позволяет его читать и менять, но при этом требует оперативную память для хранения. На основе этого универсального Java-объектного представления затем можно программно создавать объекты требуемых классов. API располагается в пакете org.w3c.dom.*

В нижеприведенном примере DOM-обработки XML-документа создается экземпляр DOM-парсера builder, метод которого buider.parse строит по входному потоку XML-документа in его представление в виде объектного дерева doc. Далее обрабатывается это представление: извлекаются все элементы с именем name, печатается содержимое их первых потомков, если они есть.

Ex : Пример DOM-анализа XML-документа

```
public void domParsing(InputStream in) throws Exception {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();
// Интерфейс Document является также узлом Node
  Document doc = builder.parse(in);
  NodeList nodes = doc.getElementsByTagName("name");
  for(int i = 0; i < nodes.getLength(); i++) {
    if (nodes.item(i).hasChildNodes()) {
      String text=nodes.item(i).getFirstChild().getNodeValue();
      System.out.println(text);
    }
  }
}
```

Ex: Пример рекурсивной печати узла DOM-дерева. Печатаются только узлы и атрибуты.

```
public void printDomTree(Node node) {
  int type = node.getNodeType();
  switch (type) {
    case Node.DOCUMENT_NODE: {
       System.out.println("<?xml version=\"1.0\" ?>");
// У узла типа Document можно получить корневой узел-элемент документа
       printDomTree(((Document)node).getDocumentElement());
       break;
    }
    case Node.ELEMENT_NODE: {
      System.out.print("<" + node.getNodeName());
      NamedNodeMap attrs = node.getAttributes();
      for (int i = 0; i < attrs.getLength(); i++) printDomTree(attrs.item(i));
      System.out.print(">");
      if (node.hasChildNodes()) {
        NodeList children = node.getChildNodes();
```

```
        for (int i = 0; i < children.getLength(); i++) printDomTree(children.item(i));
      }
      System.out.print("</" + node.getNodeName() + ">");
      break;
   }
   case Node.ATTRIBUTE_NODE: {
     System.out.print(" " + node.getNodeName() + "=\"" + ((Attr)node).getValue() + "\"");
     break;
   }
}
```

Технология StAX (Streaming API for XML, потоковое API для XML) основана на чтении XML-документа и представлении его как потока составляющих (элементы, атрибуты, текст и т.п.) или событий. При этом обработчик не вызывается парсером, как это делается в SAX, а сам читает поток представления, передающийся ему StAX-парсером. StAX-анализ позволяет читать XML-документ, а при журналировании потока накапливать текстовое представление и, следовательно, менять его. Потребляемая память зависит от типа обработки, от уровня SAX до уровня DOM.

В StAX существует 2 интерфейса: Cursor API и Event Iterator API. При помощи курсора или итератора можно передвигаться по потоку представления и обрабатывать его. API располагается в пакетах: javax.xml.stream.* и javax.xml.stream.events.*

Ниже пример StAX-анализа при помощи Cursor API. В коде создается парсер в виде потока представления reader, оборачивающего входной поток XML-документа in. Передвижение по представлению задается сдвигом курсора reader.next(). На каждом сдвиге если текущая составляющая XML-представления является началом элемента, то идет сверка его имени с шаблоном и печать содержимого в случае соответствия. По завершению работы поток представления закрывается reader.close().

Ex : Пример StAX-анализа XML-документа с использованием Cursor API

```
public void staxCursorParser(InputStream in) {
  XMLInputFactory inputFactory = XMLInputFactory.newInstance();
  XMLStreamReader reader = inputFactory.createXMLStreamReader(in);
  while(reader.hasNext()) {
    reader.next();
    if(reader.isStartElement()) {
      String qName=reader.getName().toString();
      if(qName.equals("name") || qName.equals("phone") || qName.equals("address")) {
        System.out.println(reader.getElementText());
      }
    }
  }
  reader.close();
}
```

Ниже пример StAX-анализа при помощи Event Iterator API. В коде создается парсер в

виде потока событий evr, оборачивающего входной поток XML-документа in. Передвижение по представлению задается итератором evr.nextEvent(). На очередной итерации если встречено событие начала нового элемента заданного имени, то в потоке проматываются и пишутся все последующие текстовые события. Таким образом на печать выводится содержимое всех интересующих элементов XML-документа:

Ex : Пример StAX-анализа XML-документа с использованием Event Iterator API

```
public void staxEventIteratorParser(InputStream in) {
  XMLEventReader evr = inputFactory.createXMLEventReader(in);
  while (evr.hasNext()) {
    XMLEvent event = evr.nextEvent();
    if (event.isStartElement()) {
      StartElement startElement = event.asStartElement();
      String qName = startElement.getName().toString();
      if(qName.equals("name") || qName.equals("phone") || qName.equals("address")) {
        event = evr.nextEvent();
        StringBuffer st = new StringBuffer();
        while(event.isCharacters()) {
          Characters ch = event.asCharacters();
          st.append(ch.getData());
          event = evr.nextEvent();
        }
        System.out.println(st);
      }
  }
  evr.close();
}
```

Rem: неприятная особенность реализации (JSE 8) StAX – промотка части InputStream после прочтения конца XML-документа. Также StAX ридеры не поддерживают интерфейса Closeable, что делает невозможным использование с ними конструкции try-с-ресурсами.

JSON

JSON (Java Script Object Notation) – облегченный текстовый кроссплатформенный формат обмена данными. Данные кодируются в кодировке Unicode (в реальности шире, в зависимости от реализации парсеров), с использованием представлений UTF-8, UTF-16, UTF-32. MIME-тип "application/json". Синтаксис JSON основан на литералах JavaScript (ECMA).

Данные JSON представлены примитивами и 2 структурными типами:

Примитивы:

- числа (number, integer)
- булевы значения boolean (true/false)

- строки string
- null, any

Структурные типы:
- объект (object)
- массив (array)

Объект — конструкция вида: {x1:y1, …, xN:yN} , где xi – строковый параметр (ключ), yi – значение любого типа. Пары ключ/значение хранятся произвольно (нет индекса), разделены запятыми, объект ограничен фигурными скобками.

Массив — конструкция вида: [z1, …, zN], где zj – значение любого типа, хранящегося последовательно (под индексом j). Значения массива разделены запятыми, массив ограничен квадратными скобками.

Легкость формата заключается в относительно малой доле метаинформации (по сравнению с XML) и простоте синтаксиса, дающих экономию трафика и вычислительной сложности обработки.

Ex : пример JSON-формата данных

```
{"item":
  {"id":“id3”,
    "name":{"$":"test"},
    "size":{"$":"4"},
    "info":[{"$","test info"}, {"$", "extra info"}]
  }
}
```

XML-аналог:

```
<item id=”id3”>
  <name>test</name>
  <size>4</size>
  <info>test info</info>
  <info>extra info</info>
</item>
```

В данном примере для моделирования значений XML-тегов вроде <name>test</name> использовались ”безымянные” объекты с именем ”$”. А для значений повторяющихся тегов использовался массив подобных ”безымянных” объектов.

Rem: на стороне клиента JSON-представление обычно обрабатывается JavaScript, в котором для этого есть встроенная функция eval, превращающая эти данные в JS-объект: var JSON_object= eval("("+JSON_data+")");

Структуру JSON-данных можно описать схемой, например, JSON-Schema :

Ex : пример JSON-Schema

```
{ "description":"item",
  "type":"object",
  "properties": {
    "id":{"type":"string"},
    "name":{"type":"string"},
    "size":{"type":"integer","maximum":125},
    "info":{"type":"array","items":{"type":"string"}}
  }
}
```

Java API полуавтоматической обработки JSON стандартизовано в JEE под названием Java API for JSON Processing (JSON-P, JSR 353). Находится это добро в пакетах javax.json и javax.json.*.

В JSON-P есть API парсера (разбора), строящего по JSON-данным универсальное дерево Java-объектов в памяти (аналогично DOM для XML) и располагающееся в javax.json. Также имеется и потоковый обработчик в пакете javax.json.stream (аналогично StAX для XML).

Ниже примеры методов генерации и парсинга при помощи универсального Java-объектного дерева JSON-данных определенного (без схемы) формата. Для работы парсера требуется вендорная реализация API javax.json (библиотека JEE Glassfish, например). Для экономии места описание DTO Item опущено, Items – реализация List<Item>.

Ex: пример парсинга и генерации JSON-данных

```
{
  "items": {
    "id1": {
      "name": "test1",
      "size": 4,
      "info": ["test info A", "test info B"]
    },
    "id2": {
      "name": "test2",
      "size": 4,
      "info": ["test info C"]
    }
  }
}

private static Items parsingJSON(byte[] json) throws Exception {
  Items result = new Items();
  try (InputStream in = new ByteArrayInputStream(json);
         JsonReader reader = Json.createReader(in)) {
    JsonObject root = reader.readObject();
```

```
    JsonObject jItems = root.getJsonObject("items");
    for (String id : jItems.keySet()) {
      Item item = new Item();
      item.id = id;
      JsonObject jItem = jItems.getJsonObject(id);
      item.name = jItem.getString("name");
      item.size = jItem.getInt("size");
      JsonArray jInfo = jItem.getJsonArray("info");
      List<JsonValue> list = jInfo.getValuesAs(JsonValue.class);
      for (JsonValue jv : list) {
        item.getInfo().add(jv.toString());
      }
      result.add(item);
    }
  }
  return result;
}

private static byte[] createJSON(Items items) throws Exception {
  byte[] result = null;
  try (ByteArrayOutputStream out = new ByteArrayOutputStream();
         JsonGenerator generator = Json.createGenerator(out)) {
    generator.writeStartObject().writeStartObject("items");
    for (Item item : items) {
      generator.writeStartObject(item.id).write("name", item.name).write("size",
item.size).writeStartArray("info");
      for (String str : item.info) {
        generator.write(str);
      }
      generator.writeEnd().writeEnd();
    }
    generator.writeEnd().writeEnd();
    generator.flush();
    result = out.toByteArray();
  }
  return result;
}
```

В JEE7 полностью автоматизированная обработка, т.е. связывание JSON-представления с Java-объектами все еще не стандартизовано. Предполагается, что стандартная реализация JSON Binding (JSON-B) войдет в состав одной из следующих версий JEE. Поэтому для работы с JSON-представлениями нужно подключать вендор-специфические реализации JSON-обработки вроде EclipseLink MOXy (Glassfish 4). Стандартный API JSON-B ожидается в JEE8.

Ex : Пример подключения вендор-специфических поставщиков MessageBodyReader и MessageBodyWriter (находятся в MOXyJsonProvider) для связывания с JSON:

```
@ApplicationPath("items")
public class ApplicationConfig extends Application {
```

```
  private final Set<Class<?>> classes;
  public ApplicationConfig() {
    HashSet<Class<?>> c = new HashSet<>();
    c.add(BidServiceRS.class);
    c.add(MOXyJsonProvider.class);
    classes = Collections.unmodifiableSet(c);
  }
  @Override
  public Set<Class<?>> getClasses() {
    return classes;
  }
}
```

Ругательства

- WS = Web Service — веб-служба
- SOAP = Simple Object Access Protocol (простой протокол доступа к объектам) – распределенный объектный протокол, основанный на обмене структурированными XML-сообщениями, каждое из которых представляет собою сочетание SOAP-элементов и прикладной информации.
- WSDL = WS Description Language – XML-язык описания WS.
- RPC = Remote Procedure Call – технологии удаленного вызова функций, различаются транспортным протоколом и языком сериализации/маршаллинга.
- JAX-WS = Java API for XML WS
- JAX-PRC = Java API for XML-based RPC
- SAAJ = SOAP with Attachment API for Java
- JAXR = Java API for XML Registries
- WS-I = Web Services Interoperability Organization – объединение ведущих enterprise-вендоров (Oracle, IBM, Microsoft, JBoss, HP, BEA), созданное для обеспечения межоперабельности WS для всех платформ.
- Basic Profile 1.1 = базовая конфигурация v1.1 – спецификация WS-I, описывающая набор правил по созданию межоперационно взаимодействующих WS при помощи XML, WSDL, SOAP.
- DOM = Document Open Model – абстрактный API для работы с HTML/XML документами, которые представляются в виде древообразной структуры данных. Описывается спекой W3C DOM. Реализован во всех современных браузерах. JDOM – Java-реализация DOM.
- SAX = Simple API for XML – абстрактный API для последовательного чтения XML-документов (что-то вроде стековой обработки постфиксной записи выражений). При этом объем используемой при разборе памяти мал и фиксирован, однако разбираемый документ трудно изменить. Используется для чтения XML и построения DOM-модели. Реализации парсеров (в т.ч. Apache Xerces) используют событийную модель разбора. События, генерируемые при парсинге, обрабатываются обработчиками обратного вызова (callback handler), регистрируемыми перед началом обработки.
- XML = eXtensible Markup Language = расширяемый язык разметки.

- XML Schema – язык описания структуры XML-документов, т.е. язык для написания языков разметки.
- XSD = XML Schema Definition = описание схемы XML, т.е. определение конкретного разметочного языка для соответствующих этой схеме XML-документов. Само XSD является XML-документом.
- XSD-документ = XSD, рассматриваемый как XML-документ.
- XML Namespace = пространство имен XML — уникальный идентификатор конкретного языка XML-разметки, заданного XSD. Состоит из префикса (короткого локального имени, уникального для использующего его XML-документа) и URI – идентификатора, уникального для всей области использования этого языка разметки.
- XSI = XML Schema Instance – язык разметки, содержащий атрибуты для более полного применения XSD в XML-документах. URI схемы XSI ”http://www.w3.org/2001/XMLSchema-Instance”. В качестве префикса обычно используют “xsi”.
- XS = XML Schema – основная схема XSD с URI – “http://www.w3.org/2001/XMLSchema”. В качестве префикса обычно используют “xs” или “xsd”, либо объявляют ее пространством имен по-умолчанию.
- XS-типы = встроенные типы XML – базовые типы данных, определенные в XS.
- QName = F.Q.N. = полностью квалифицированное имя Prefix:LocalPart . QName может и не содержать префикса, если гарантируется локальная уникальность, например, для атрибутов или для элементов из пространства имен по-умолчанию.
- UDDI = Universal Description Discovery & Integration (универсальное описание, обнаружение и встраивание) — стандарт, описывающий обнаружение и публикацию WS в сети.
- UBR = Universal Business Registry – универсальный бизнес-реестр.
- SAAJ — API управления структурой SOAP-сообщений
- JAXR — API получения доступа к реестрам WS, обычно UDDI
- DII = Dynamic Invocation Interface — динамический интерфейс вызова
- Generated Stubs = генерируемые заглушки
- Dynamic Proxy = динамический прокси — посредник в виде Java-класса, который создается в deploy-time на основе WSDL, реализует интерфейс конечной точки WS и является посредником между клиентом и WS
- WS Endpoint (в общем) = URL, по которому WS может быть запрошен клиентом, каждый WS может иметь несколько Endpoint (разные протоколы, разные привелегии и т.п.)
- SEI = Service Endpoint Interface — интерфейс конечной точки WS – представление конечной точки веб-службы в виде @WebService Java-интерфейса, в котором перечисляются все методы, предлагаемые клиентам в качестве операций WS. SEI связывается с SEIC посредством метаинформации.
- SEIC = Service Endpoint Implementation Class = класс реализации конечной точки WS – Java-реализация конечной точки веб-службы в виде аннотированного @WebService или @WebServiceProvider класса.
- JAX-WS Endpoint = Java-представление конечной точки WS в JAX-WS, т.е. либо SEI, либо SEIC.

- EJB Endpoint = конечная точка EJB – Java-представление EJB-контейнерный вариант SEIC, SLSB или @Singleton.
- Servlet Endpoint = конечная точка сервлета — POJO-вариант SEIC для Web-модели
- Service Class = Service-класс = класс веб-службы — класс для для связывания клиента с WS.
- JAXB = Java Architecture for XML Binding - Java-архитектура для связывания с XML. Описываются спецификациями JSR-222.
- xjc = XML to Java Compiler – компилятор Java-классов из XSD для JAXB. Входит в состав JDK начиная с JSE6.
- wsgen = WSDL Generator – генератор WSDL-документации из JAX-WS веб-службы (SEI - класса). Входит в состав JDK с JSE 6.
- wsimport = WSDL Import – инструмент развертывания JAX-WS на основе WSDL-документации. Входит в состав JDK с JSE 6.
- JAXP = Java API for XML Processing – набор фреймворков для отображения XML на Java-объекты. Включает SAX – API событийного разбора XML-документов, DOM – API построения объектного дерева XML и StAX – API потокового разбора XML-документов.
- WS contract – контракт, заключаемый между потребителем (клиентом) и поставщиком (веб-службой), для SOAP это WSDL-документ.
- Компонент JAX-WS – реализация веб-службы JAX-WS в рамках JAX-WS-совместимого сервера, состоящая из WSDL, SEIC, SEI, обработчиков, системного прокси (сервлета).
- CRUD = Create, Read, Update, Delete
- MIME = Multipropose Internet Mail Extension = многоцелевые расширения интернет-почты
- REST = Representational State Transfer = передача репрезентативного состояния
- Ресурс RESTful – любое нечто, которое клиент может запросить по ссылке или с которым он может взаимодействовать. Т.е. это все, что заслуживает отдельной ссылки.
- Представление ресурса — любая информация об его состоянии, различают человеко-читаемые и машинно-обрабатываемые представления
- Репрезентативность состояния означает, что одно и то же состояние ресурса может быть передано в различных представлениях.
- JAX-RS = Java API for RESTful WS
- HTTP-объект — тело HTTP-сообщения
- POJO = Plain Old Java Object = старый добрый Java-объект — объект, который не расширяет и не реализует специфические типы, не входящие в бизнес-логику приложения.
- Текучий интерфейс = fluent interface — название API, реализующего цепочку методов. В цепочке каждый метод должен возвращать некий общий контекст, который позволяет делать дальнейшие вызовы (в Java через точку). Тем самым в одной строке компактно можно осуществить многократный вызов методов и реализовать разнообразные сценарии формирования объекта, поддерживающего данный API.

Литература

Enterprise JavaBeans 3.1 (6th edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Куправк, Майкл Ремижан, стр. nnn

Java EE 6 и сервер приложений GlassFish v3, Дэвид Хэффельфингер

Веб-сервисы Java. Тимур Машанин 2012

http://examples.oreilly.com/9780596002527/namespaces.html

https://www.w3.org/2009/XMLSchema/XMLSchema.xsd

https://www.ibm.com/developerworks/library/ws-usagewsdl/

http://www.tutorialspoint.com/wsdl/wsdl_message.htm

https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/cwbs_jaxwsclients.html

http://docs.oracle.com/cd/E21764_01/web.1111/e13734/provider.htm#WSADV582

https://docs.oracle.com/cd/E26576_01/doc.312/e24929/dd-files.htm#GSDPG00078

http://learningviacode.blogspot.ru/2013/10/soap-action-header.html

http://www.javaportal.ru/java/articles/java_http_web/article03.html

http://www.javaportal.ru/java/articles/java_http_web/article03.html

http://blog.bdoughan.com/2013/06/moxy-is-new-default-json-binding.html