Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 13. Компонентные технологии разработки Web-приложений

Аннотация

Рассматриваются основные элементы компонентных сред Java 2 Enterprise Edition и .NET. Показывается, как в рамках этих технологий решаются основные задачи построения распределенных Web-приложений.

Ключевые слова

Web-приложения, расширяемый язык разметки XML, XSLT, схема документа XML, Web-контейнер, EJB-контейнер, Web-компоненты J2EE, компоненты EJB, дескрипторы развертывания, Java RMI, JMS, JNDI, JTA, защита на основе ролей, DOM, SAX, StAX, ADO.NET, ASP.NET, конфигурационные файлы .NET, .NET Remoting, Active Directory, зоны приложений

Текст лекции

После обзора общих концепций, связанных с компонентными технологиями и распределенным программным обеспечением, отметим дополнительные общие черты современных компонентных технологий.

Программное обеспечение в современном мире становится все сложнее и приобретает все больше функций. Коммерческие компании и государственные организации стремятся автоматизировать все больше своих процессов, как внутренних, так и тех, что связаны с общением с внешним миром. При этом, однако, разработка таких приложений, их внедрение и поддержка становятся все дороже.

Есть, тем не менее, фактор, который помогает значительно снизить расходы на внедрение и поддержку — широчайшее распространение Интернет. Если ваше программное обеспечение использует для связи между своими элементами базовые протоколы Интернет (TCP/IP и HTTP) и предоставляет пользовательский интерфейс с помощью HTML, который можно просматривать в любом браузере, то практически каждый его потенциальный пользователь не имеет технических препятствий для обращения к этому ПО. Не нужно распространять специальные клиентские компоненты, ставить клиентам специальное оборудование, не нужно тратить много времени и средств на обучение пользователей работе со специфическим интерфейсом, настройке связи с серверами и т.д. Интернет предоставляет готовую инфраструктуру для создания крупномасштабных программных систем, в рамках которых десятки тысяч компонентов могли бы работать совместно и миллионы пользователей могли бы получать услуги.

Поэтому вполне логично, что ***Web-приложения***, т.е. программные системы, использующие для связи протоколы Интернет, а в качестве пользовательского интерфейса HTML страницы, стали одним из самых востребованных видов ПО. Однако, чтобы сделать потенциальные выгоды от использования Интернет реальными, необходимы технологии разработки Web-приложений, которые позволяли бы строить их на компонентной основе, минимизируя затраты на интеграцию отдельных компонентов, их развертывание и поддержку в рабочем состоянии.

Другим важным фактором является распространение ***расширяемого языка разметки (Extensible Markup Language, XML)*** как практически универсального формата данных. XML предоставляет стандартную лексическую форму для представления текстовой

информации различной структуры и стандартные же способы описания этой структуры. Многие аспекты создания и работы Web-приложений связаны с обменом разнообразно структурированными данными между отдельными компонентами или представлением информации об организации, свойствах и конфигурации системы, имеющей гибкую структуризацию. Использование XML позволяет решить часть возникающих здесь проблем.

Поскольку все современные технологии разработки Web-приложений так или иначе используют XML, следующий раздел посвящен основным элементам этого языка.

Расширяемый язык разметки XML

Данный раздел содержит краткий обзор основных конструкций XML, для более глубоко изучения этого языка и связанных с ним технологий см. [1-7].

XML [3-5] является *языком разметки*: различные элементы данных в рамках XML-документов выделяются *тегами* — каждый элемент начинается с открывающего тега `<tag>` и заканчивается закрывающим `</tag>`. Здесь `tag` — идентификатор тега, который обычно является английским словом или набором слов, разделяемых знаками ‘-’, обозначающим назначение этого элемента данных. Элементы данных могут быть вложены друг в друга, образуя дерево документа. Кроме того, каждый элемент может иметь набор значений атрибутов, которые представляют собой строки, числа или логические значения. Значения атрибутов для данного элемента помещаются внутри его открывающего тега. Элемент данных, не имеющий вложенных подэлементов, может быть оформлен в виде конструкции `<tag … />`, т.е. не иметь отдельного закрывающего тега.

Ниже приведен пример описания информации о книге в виде XML.

```
<book
    title = "Pattern-Oriented Software Architecture, Volume 1: A System of
Patterns"
    ISBN = "047195869"
    year = 1996
    hardcover = true
    pages = 476
    language = "English">
  <author>Frank Buschmann</author>
  <author>Regine Meunier</author>
  <author>Hans Rohnert</author>
  <author>Peter Sommerlad</author>
  <author>Michael Stal</author>
  <publisher
      title = "John Wiley & Sons"
      address = "605 Third Avenue, New York, NY 10158-0012, USA" />
</book>
```

В этом примере тег `<book>`, представляющий описание книги, имеет вложенные теги `<author>` и `<publisher>`, представляющие ее авторов (таких тегов может быть несколько) и издателя, а также атрибуты `title`, `ISBN`, `year`, `hardcover`, `pages` и `language` (название книги, ее международный стандартный номер, т.е. International Standard Book Number, ISBN, а также год издания, наличие твердой обложки, число страниц и язык). Тег `<publisher>`, в свою очередь, имеет атрибуты `title` и `address` (название и юридический адрес издательской организации).

Элементы XML-документа, называемые также *сущностями*, являются в некотором смысле аналогами значений структурных типов в .NET, а значения их атрибутов — аналогами соответствующих значений полей. При этом теги играют роль самих типов, а атрибуты и вложенные теги — роль их полей, имеющих, соответственно, примитивные и структурные типы. Расширяемым XML назван потому, что можно задать специальную структуру тегов и их атрибутов для некоторого вида документов. Эта структура описывается в отдельном документе, называемом *схемой*, который сам написан на

специальном подмножестве XML, *DTD (Document Type Declaration, декларация типа документа)* [3-5] или *XMLSchema* [6].

XML-документ всегда начинается заголовком, описывающим версию XML, которой соответствует документ, и используемую кодировку. По умолчанию используется кодировка UTF-8.

Затем чаще всего идет описание типа документа, указывающее схему, которой он соответствует, и тег корневого элемента, содержащего все остальные элементы данного документа. Схема может задаваться в формате DTD или XMLSchema. Второй, хотя и является более новым, пока еще используется реже, потому что достаточно много документов определяется с помощью DTD и очень многие инструменты для обработки XML могут пользоваться этим форматом. Используемая схема определяется сразу двумя способами — при помощи строки, которая может служить ключом для поиска схемы на данной машине, и при помощи унифицированного идентификатора документа (Unified Resource Identifier, URI), содержащего ее описание, и используемого в том случае, если ее не удалось найти локально.

Ниже приводится пример заголовка и описания типа документа для дескриптора развертывания EJB компонентов (см. подробности далее).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application
Server 8.1 EJB 2.1//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-
jar_2_1-1.dtd">
<sun-ejb-jar>
…
</sun-ejb-jar>
```

Другой пример показывает заголовок документа DocBook — основанного на XML формата для технической документации, которая может быть автоматически преобразована в HTML, PDF и другие документы с определенными для них правилами верстки.

```
<?xml version="1.0" encoding="windows-1251"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article>
…
</article>
```

Помимо элементов данных и заголовка с описанием типа документа, XML-документ может содержать комментарии, помещаемые в теги `<!-- … -->`, *инструкции обработки* вида `<? processor-name … ?>` (здесь `processor-name` — идентификатор обработчика, которому предназначена инструкция) и *секции символьных данных* CDATA, которые начинаются набором символов `<![CDATA[`, а заканчиваются с помощью `]]>`. Внутри секций символьных данных могут быть любые символы, за исключением закрывающей комбинации. В остальных местах некоторые специальные символы должны быть представлены комбинациями символов в соответствии с Таблицей 1.

Символ	Представление в XML
<	<
>	>
&	&
"	"
'	'

Таблица 1. Представления специальных символов в XML.

XML содержит много других конструкций, помимо уже перечисленных, но их рассмотрение выходит за рамки данного курса. Читатель, желающий узнать больше об этом языке и связанных с ним технологиях, может обратиться к [1-7].

Платформа Java 2 Enterprise Edition

Платформа J2EE предназначена в первую очередь для разработки распределенных Web-приложений и поддерживает следующие 4 вида компонентов [8].

- ***Enterprise JavaBeans (EJB).***
  Компоненты EJB предназначены для реализации на их основе бизнес-логики приложения и операций над данными. Любые компоненты, разработанные на Java, принято называть бинами (bean, боб или фасолина, в разговорном языке имеет также значения головы и монеты). Компоненты Enterprise JavaBean отличаются от «обычных» тем, что работают в рамках ***EJB-контейнера***, который является для них компонентной средой. Он поддерживает следующие базовые службы при работе с компонентами EJB.
    - Автоматическую поддержку обращений к компонентам, размещенным на разных машинах.
    - Автоматическую поддержку транзакций.
    - Автоматическую синхронизацию состояния баз данных и соответствующих компонентов EJB в обе стороны.
    - Автоматическую поддержку защищенности за счет аутентификации пользователей, проверки прав пользователей или компонентов на выполнение выполняемых ими операций и авторизации производимых действий.
    - Автоматическое управление жизненным циклом компонента (последовательностью переходов между состояниями типа «отсутствует»-«инициализирован»-«активен») и набором компонентов как ресурсами: удаление компонентов, ставших ненужными; загрузку новых компонентов; балансировку нагрузки между имеющимися компонентами; использование пула готовых к работе, но не инициализированных компонентов, чтобы не тратить время на их удаление и создание, и пр.

  В целом EJB-контейнер представляет собой пример ***объектного монитора транзакций (object transaction monitor)*** — ПО промежуточного уровня, поддерживающего в рамках объектно-ориентированной парадигмы удаленные вызовы методов и распределенные транзакции.
- ***Web-компоненты (Web components).***
  Эти компоненты служат для предоставления интерфейса к корпоративным программным системам поверх широко используемых протоколов Интернет, а именно, HTTP. Предоставляемые интерфейсы могут быть как интерфейсами для людей (WebUI), так и специализированными программными интерфейсами, работающими подобно удаленному вызову методов, но поверх HTTP.
  В группу Web-компонентов входят *фильтры (filters)*, *обработчики Web-событий (web event listeners)*, ***сервлеты (servlets)*** и ***серверные страницы Java (JavaServer Pages, JSP)***.
  Компонентной средой для работы Web-компонентов служит ***Web-контейнер***, поставляемый в рамках любой реализации платформы J2EE. Web-контейнер реализует такие службы, как управление жизненным циклом компонентов, управление набором компонентов как ресурсом, распараллеливание независимых работ, выполнение удаленных обращений к компонентам, поддержка

защищенности с помощью проверки прав компонентов и пользователей на выполнение различных операций.

- Обычные приложения на Java.
  J2EE является расширением J2SE и поэтому все Java приложения могут работать и в этой среде. Однако, в дополнение к обычным возможностям J2SE, эти приложения могут использовать в своей работе Web-компоненты и EJB, как напрямую, так и удаленно, связываясь с ними по HTTP.
- Аплеты (applets).
  Это небольшие компоненты, имеющие графический интерфейс пользователя и предназначенные для работы внутри стандартного Web-браузера. Они используются в тех случаях, когда не хватает выразительных возможностей пользовательского интерфейса на базе HTML и могут связываться с удаленными, работающими на сервере Web-компонентами по HTTP.

Компонент любого из этих видов оформляется как небольшой набор классов и интерфейсов на Java, а также имеет *дескриптор развертывания (deployment descriptor)* — описание в определенном формате на основе XML конфигурации компонента в рамках контейнера, в который он помещается. Приложение в целом также имеет дескриптор развертывания. Дескрипторы развертывания играют важную роль, позволяя менять некоторые параметры функционирования компонента и привязывать их к параметрам среды, в рамках которой компонент работает, не затрагивая его код.

Платформа J2EE приспособлена для разработки многоуровневых Web-приложений. При работе с такими приложениями пользователь формирует свои запросы, заполняя HTML-формы в браузере, который упаковывает их в HTTP-сообщения и пересылает Web-серверу. Web-сервер передает эти сообщения Web-компонентам, выделяющим из них исходные запросы пользователя и передающим их для обработки компонентам EJB. Результаты работы EJB компонентов превращаются Web-компонентами в динамически генерируемые HTML-страницы, и отправляются обратно пользователю, представая перед ним в окне браузера. Аплеты используются там, где нужен более функциональный интерфейс, чем стандартные формы и страницы HTML.

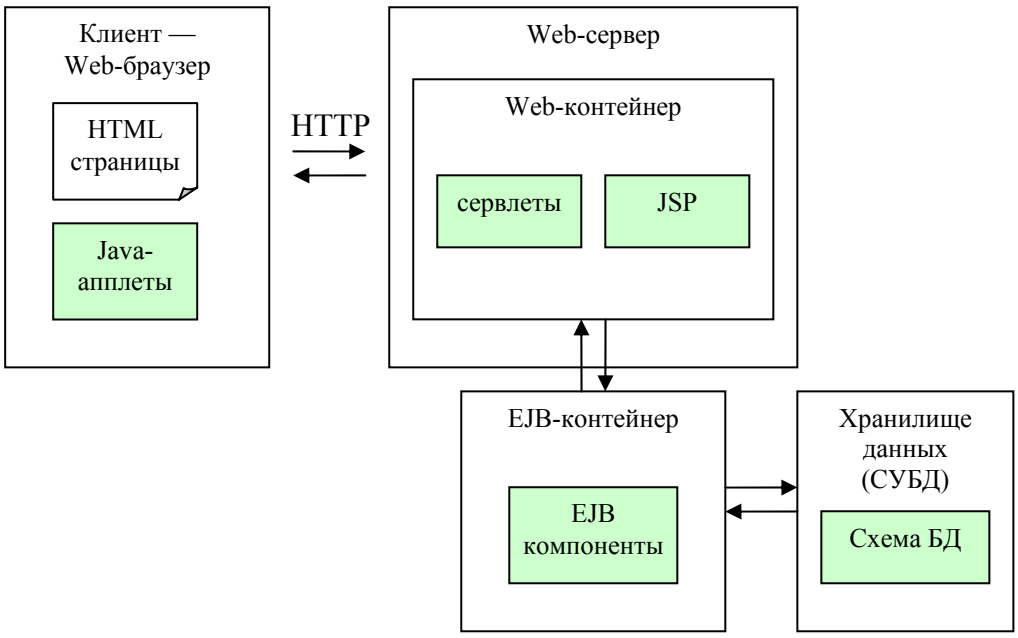

Рисунок 1. Типовая архитектура J2EE приложения.
Выделены компоненты, разрабатываемые вручную.

Таким образом, приложения на базе J2EE строятся с использованием трех основных архитектурных стилей.

- *Многоуровневая система.*
  Самые крупные подсистемы организованы как уровни, решающие различные задачи.
  - Интерфейс взаимодействия с внешней средой, включая пользователей, реализуется при помощи Web-компонентов.
  - Уровень бизнес-логики и модели данных реализуется при помощи EJB компонентов.
  - Уровень управления ресурсами строится на основе коммерческих систем управления базами данных (СУБД). Можно также подключать другие виды ресурсов, для которых имеется реализация интерфейса поставщика служб J2EE (J2EE service provider interface, J2EE SPI).
- *Независимые компоненты.*
  Первые два уровня построены из отдельных компонентов, каждый из которых имеет собственную область ответственности, но может привлекать для решения частных задач другие компоненты.
- *Данные-представление-обработка (MVC).*
  Работа компонентов в рамках обработки группы тесно связанных запросов организуется по образцу MVC. При этом сервлеты и обработчики Web-событий служат обработчиками, компоненты JSP — представлением, а компоненты EJB — моделью данных.

Рассмотрим теперь, как решаются общие задачи построения распределенных систем [9] на базе платформы J2EE.

Связь

Связь между компонентами, работающими в разных процессах и на разных машинах, обеспечивается в J2EE, в основном, двумя способами: синхронная связь — с помощью реализации удаленного вызова методов на Java (*Java RMI*), асинхронная — с помощью *службы сообщений Java (Java message service, JMS)*.

Java RMI [10] является примером реализации общей техники удаленного вызова методов. Базовые интерфейсы для реализации удаленного вызова методов на Java находятся в пакете `java.rmi` стандартной библиотеки.

Набор методов некоторого класса, доступных для удаленных вызовов, выделяется в специальный интерфейс, называемый ***удаленным интерфейсом (remote interface)*** и наследующий `java.rmi.Remote`. Сам класс, методы объектов которого можно вызвать удаленно, должен реализовывать этот интерфейс. Этот же интерфейс реализуется автоматически создаваемой клиентской заглушкой. Поэтому объекты-клиенты работают только с объектом этого интерфейса, а не с объектом класса, реализующего декларированные в таком интерфейсе операции.

Кроме того, класс, реализующий удаленный интерфейс, должен наследовать классу `java.rmi.server.RemoteObject`. Этот класс содержит реализации методов `hashCode()`, `equals()` и `toString()`, учитывающие возможность размещения его объектов в процессах, отличных от того, где они вызываются. Обычно наследуется не сам этот класс, а его подклассы `java.rmi.server.UnicastRemoteObject` или `java.rmi.activation.Activatable`. Первый реализует общую функциональность объектов, которые можно вызвать удаленно поверх транспортного протокола TCP, пока работает содержащий их процесс, включая занесение информации о таких объектах в реестр RMI (собственная служба именования в рамках Java RMI). Второй класс служит для реализации ***активизируемых объектов (activatable objects)***, которые создаются

системой «по требованию» — как только кто-нибудь вызывает в них какой-нибудь метод. Ссылки на такие объекты могут сохраняться, а обратиться к ним можно спустя долгое время, даже после перезагрузки системы.

Каждый ***удаленный метод (remote method)***, т.е. метод, который можно вызвать из другого процесса, должен декларировать в качестве класса возможных исключений `java.rmi.RemoteException` или его базовые типы `java.io.IOException` или `java.lang.Exception`. Этот класс сам служит базовым для классов исключений, представляющих ошибки связи, ошибки маршалинга параметров или результатов и ошибки протоколов, реализующих RMI.

Объекты, реализующие один из удаленных интерфейсов, могут быть переданы в качестве параметров или результатов удаленных методов по ссылке как объекты этого интерфейса. При этом в другой процесс передается байт-код клиентской заглушки, связанной с таким объектом, и объекты этого процесса получают возможность вызывать в нем методы.

Остальные объекты передаются как параметры или результаты удаленных вызовов с помощью сериализации их данных и построения копии такого объекта в другом процессе. Так же передаются и создаваемые удаленным методом исключения. Для этого им необходимо реализовывать интерфейс `java.io.Serializable` или же быть значениями примитивных типов.

Простой пример Java классов, взаимодействующих по RMI, приведен ниже.

Код удаленного интерфейса.

```
package examples.rmi;

public interface Hello extends java.rmi.Remote
{
  public String hello() throws java.rmi.RemoteException;
}
```

Код реализации удаленного интерфейса.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
  public static final String ServerHost = "hostname";
  public static final String ServerURL = "rmi://" + ServerHost
                                         + ":2001/SERVER";
  public static final int RegistryPort = 2000;

  public HelloImpl () throws java.rmi.RemoteException { }

  public String hello () throws java.rmi.RemoteException
  { return "Hello!"; }

  public static void main (String[] args)
  {
    try
    {
      Hello stub = new HelloImpl();
      Registry registry = LocateRegistry.getRegistry(RegistryPort);
      registry.rebind(ServerURL, stub);
    }
    catch (Exception e)
    {
      System.out.println("server creation exception");
```

```
      e.printStackTrace();
    }
  }
}
```

Код класса-клиента.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient
{
  public HelloClient () { }

  public static void main (String[] args)
  {
    try
    {
      Registry registry = LocateRegistry.getRegistry
                          (HelloImpl.ServerHost, HelloImpl.RegistryPort);
      Hello stub = (Hello) registry.lookup(HelloImpl.ServerURL);
      System.out.println("response: " + stub.hello());
    }
    catch (Exception e)
    {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

Для того чтобы запустить этот пример, нужно выполнить следующие действия.

- Скомпилировать все классы и интерфейсы. В классе-реализации серверного компонента константа `ServerHost` должна быть инициализирована именем машины, на которой будет работать сервер.
- Создать и скомпилировать заглушки при помощи компилятора Java RMI, запустив его в корневой директории для кода на Java.
  `rmic examples.rmi.HelloImpl`
- Запустить реестр RMI на той машине, на которой будет выполняться серверный компонент `HelloImpl`. Реестр используется для регистрации серверного объекта и последующего поиска его клиентами и в данном примере принимает сообщения по порту 2000.
  `start rmiregistry 2000 &`
- Запустить сам серверный компонент.
  `java examples.rmi.HelloImpl`
- Запустить клиентский компонент на той же машине или на другой.
  `java examples.rmi.HelloClient`
  Если никаких ошибок не сделано, и порты 2000 и 2001 на серверной машине свободны, клиент выдаст сообщение.
  `response: Hello!`

Поскольку базовые интерфейсы компонентов EJB наследуют `java.rmi.Remote`, такие компоненты автоматически предоставляют возможность обращаться к некоторым из своих методов удаленно.

Служба сообщений Java (JMS) [11] представляет собой спецификацию интерфейсов для поддержки передачи сообщений (в частности, асинхронных) между компонентами в рамках платформы J2EE. Ее базовые библиотеки — пакет `javax.jms` — не входят в состав J2SE. Та или иная реализация JMS должна входить в любую реализацию платформы J2EE.

Основными элементами JMS являются следующие.

- *Сообщение*. Все объекты-сообщения реализуют интерфейс `javax.jms.Message`. Сообщение имеет тело, которое может быть отображением (map) ключей в значения, текстом, объектом, набором байт или потоком значений примитивных типов Java. Каждый из видов сообщений представлен особым подинтерфейсом общего интерфейса `Message`.
  Сообщение может иметь набор заголовков (headers), большинство из которых определяются автоматически.
  Кроме того, сообщение может иметь набор свойств, которые позволяют определять дополнительные заголовки, специфичные для данного приложения.
- Клиент может передать сообщение, установив *соединение* с провайдеров JMS. Соединения представляются с помощью объектов интерфейса `javax.jms.Connection`, а получить соединение можно с помощью фабрики соединений (объект `javax.jms.ConnectionFactory`), найдя ее при помощи службы именования. Передать сообщение можно и воспользовавшись *объектом-адресатом* (объект `javax.jms.Destination`), который также можно получить через службу именования.
- JMS поддерживает как связь *точка-точка (peer-to-peer, P2P)*, так и посылку и прием сообщений по схеме *подписки/публикации*. Основные интерфейсы JMS (соединение, фабрика соединений, адресат, сессия и пр.) имеют специфические подинтерфейсы для каждой из этих двух моделей связи. В клиентских программах рекомендуется всегда использовать общие интерфейсы.

Именование

Поиск ресурсов по именам или идентификаторам и набору их свойств в рамках J2EE и J2SE осуществляется при помощи интерфейса JNDI (Java Naming and Directory Interface, Java интерфейс служб имен и каталогов) [12].

Интерфейсы и классы JNDI находятся в пакете `javax.namimg` и его подпакетах `javax.naming.directory, javax.naming.event, javax.naming.ldap`.

Основные сущности служб именования и каталогов, хранящие привязку ресурсов к именам и наборам атрибутов, называются *контекстами*. Они представляются объектами интерфейса `javax.naming.Context`, в частности, реализующих его классов `javax.naming.InitialContext, javax.naming.directory.InitialDirContext, javax.naming.ldap.InitialLdapContext`.

Методы этого интерфейса служат для привязки объекта к имени (`void bind(String, Object)`, `void rebind(String, Object)` — связать данное имя с данным объектом, даже если оно уже имеется в этом контексте), поиска объекта по имени (`Object lookup (String)`), разрыва связи между именем и объектом (`void unbind(String)`) и пр.

В дополнение к этим методам классы `InitialDirContext` и `InitialLdapContext` реализуют интерфейс контекста службы каталогов `DirContext`. Контекст службы каталогов имеет методы `void bind(String, Object, Attributes)` для привязки набора атрибутов к объекту, `Attributes getAttributes(String)` для получения набора атрибутов объекта по указанному имени и `NamingEnumeration<SearchResult> search(String, Attributes)` для поиска объектов по указанному набору атрибутов в контексте с указанным именем.

Класс `InitialLdapContext` реализует общий протокол работы со службами каталогов — *простой протокол доступа к службам каталогов (Lightweight Directory Access Protocol, LDAP)*.

При загрузке виртуальной машины механизм инициализации JNDI конструирует начальный контекст по JNDI свойствам, задаваемым во всех файлах с именем `jndi.properties`, находящихся в директориях, перечисленных в `classpath`.

Стандартный набор JNDI свойств, которые могут быть установлены для Java приложения или аплета, включает следующие:

- `java.naming.factory.initial` (соответствует константе `Context.INITIAL_CONTEXT_FACTORY`) — имя класса фабрики для создания начальных контекстов, обязательно должно быть установлено;
- `java.naming.provider.url` (соответствует константе `Context.PROVIDER_URL`) — URL сервера каталогов или имен;
- `java.naming.dns.url` (соответствует константе `Context.DNS_URL`) — URL для определения DNS узла, используемого для получения адреса JNDI URL;
- `java.naming.applet` (соответствует константе `Context.APPLET`) — объект-апплет, используемый для получения JNDI свойств;
- `java.naming.language` (соответствует константе `Context.LANGUAGE`) — список, через запятую, предпочтительных языков для использования в данной службе (пример: `en-US, fr, ja-JP-kanji`). Языки описываются в соответствии с RFC 1766 [13].

Ниже приведен пример использования JNDI для распечатки содержимого директории c:/tmp. Для работы с файловой системой через JNDI используется реализация службы именования на основе файловой системы от Sun [14].

```
package examples.jndi;

import java.util.Properties;

import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

public class JNDIExample
{
  public static void main (String[] args)
  {
    Properties env = new Properties();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file://c:/tmp");

    try
    {
      Context cntx = new InitialContext(env);
      NamingEnumeration list = cntx.listBindings("");

      while(list.hasMore())
      {
        Binding bnd = (Binding)list.next();
        System.out.println(bnd.getName());
      }
    }
    catch (NamingException e)
    {
      e.printStackTrace();
    }
  }
```

```
}
```

Процессы и синхронизация

Разбиение J2EE приложения на набор взаимодействующих процессов и потоков и управление ими осуществляется их Web- и EJB-контейнерами автоматически. На их работу можно влиять с помощью конфигурирования J2EE-сервера в целом и конкретных приложений.

Все методы вспомогательных классов, которые используются Web-компонентами или компонентами EJB, нужно объявлять синхронизированными (**`synchronized`**).

Компоненты J2EE-приложений, работающие в рамках контейнеров, могут создавать собственные отдельные потоки, но делать это нужно с большой осторожностью, поскольку этими потоками контейнер управлять не сможет, и они могут помешать работе других компонентов.

Целостность

Целостность и непротиворечивость данных при работе J2EE приложений поддерживается с помощью механизма распределенных транзакций. Управление такими транзакциями может быть возложено на EJB-контейнер, что делается с помощью определения политик участия методов EJB-компонентов в транзакциях в их дескрипторах развертывания, или может осуществляться вручную. В обоих случаях используются механизмы, реализующие *интерфейсы управления транзакциями Java (Java Transaction API, JTA)*.

Базовые интерфейсы JTA находятся в пакетах `javax.transaction`. и `javax.transaction.xa`. Это, прежде всего, интерфейсы менеджера транзакций `TransactionManager`, самих транзакций `Transaction` и `UserTransaction` и интерфейс синхронизации `Synchronization`, позволяющий получать уведомление о начале завершения и конце завершения транзакций.

Методы интерфейса `TransactionManager` позволяют стартовать транзакцию, завершить ее успешно или откатить, а также получить объект, представляющий текущую транзакцию и имеющий тип `Transaction`. Методы интерфейса `Transaction` позволяют завершить или откатить транзакцию, представляемую объектом такого интерфейса, зарегистрировать объекты для синхронизации при завершении транзакции, а также добавить некоторые ресурсы в число участников данной транзакции или удалить их из транзакции. Такие ресурсы представляются в виде объектов интерфейса `javax.transaction.xa.XAResource`. Интерфейс `UserTransaction` предназначен для управления пользовательскими транзакциями — он предоставляет немного меньше возможностей, чем `TransactionManager`.

В том случае, если управление транзакциями целиком поручается EJB-контейнеру (это так называемые *транзакции, управляемые контейнером, container managed transactions*), влиять на их ход можно, указывая в дескрипторах развертывания EJB-компонентов различные *транзакционные атрибуты (transaction attributes)* для их методов. Транзакционный атрибут может принимать одно из следующих значений.

- Required
  Метод, имеющий такой атрибут, всегда должен выполняться в контексте транзакции. Он будет работать в контексте той же транзакции, в которой работал вызвавший его метод, а если он был вызван вне контекста транзакции, с началом его работы будет запущена новая транзакция.
  Этот атрибут используется наиболее часто.
- RequiresNew
  Метод, имеющий такой атрибут, всегда будет запускать новую транзакцию в

самом начале работы. При этом внешняя транзакция, если она была, будет временно приостановлена.

- Mandatory
  Метод, имеющий такой атрибут, должен вызываться только из транзакции, в контексте которой он и продолжит работать. При вызове такого метода извне транзакции, будет создана исключительная ситуация типа `TransactionRequiredException`.
- NotSupported
  При вызове такого метода внешняя транзакция, если она есть, будет временно приостановлена. Если ее нет, новая транзакция не будет запущена.
- Supports
  Такой метод работает внутри транзакции, если его вызвали из ее контекста, если же он был вызван вне транзакции, новая транзакция не запускается.
- Never
  При вызове такого метода из транзакции создается исключительная ситуация типа `RemoteException`. Он может работать, только будучи вызван извне транзакции.

Откатить автоматически управляемую транзакцию можно, создав исключительную ситуацию типа `javax.ejb.EJBException` или вызвав метод `setRollbackOnly()` интерфейса `javax.ejb.EJBContext`.

Отказоустойчивость

Отказоустойчивость J2EE приложений не обеспечивается дополнительными средствами, такими, как репликация, надежная передача сообщений и пр. Если в них возникает необходимость, разработчик должен сам реализовывать эти механизмы, либо пользоваться готовыми решениями за рамками платформы J2EE.

Защита

Защищенность J2EE приложения поддерживается несколькими способами.

- С помощью определения методов *аутентификации*, т.е. определения идентичности пользователей. Эти методы определяются в дескрипторе развертывания приложения. Можно использовать следующие способы аутентификации.
  - Отсутствие аутентификации.
  - С помощью базового механизма протокола HTTP. При этом при попытке обращения к ресурсу по протоколу HTTP будет запрошено имя пользователя и пароль, которые будут проверены Web-сервером. Этот способ не слишком хорошо защищен, поскольку реквизиты пользователя пересылаются по сети в незашифрованном виде.
  - С помощью дайджеста (digest). Этот метод работает так же, как базовый механизм аутентификации по HTTP, но имя и пароль пользователя пересылаются в зашифрованном виде. Такой способ используется достаточно редко.
  - С помощью специальной формы. При этом определяется страница, на которой расположена форма аутентификации (обычно это те же поля для ввода имени пользователя и пароля, но, может быть, и каких-то других его атрибутов), и страница, на которой находится сообщение, выдаваемое при неудачной аутентификации.

    - С использованием сертификата клиента. Этот метод требует использовать протокол HTTPS. Клиент должен предоставить свой сертификат или открытый ключ, удовлетворяющий стандарту X.509 на инфраструктуру открытых ключей. Можно использовать и взаимную аутентификацию — в этом случае и клиент, и сервер предоставляют свои сертификаты.
- С помощью соединений по протоколу HTTP поверх уровня защищенных сокетов (Secure Socket Layer, SSL, на HTTP поверх SSL часто ссылаются с помощью отдельной аббревиатуры HTTPS).
  Можно потребовать использовать только такие соединения, указав атрибуты CONFIDENTIAL и/или INTEGRAL в дескрипторе развертывания приложения. Первый атрибут означает, что данные, передаваемые между клиентом и приложением, будут зашифрованы, так что их тяжело будет прочитать третьей стороне. Второй атрибут означает, что эти данные будут сопровождаться дополнительной информацией, гарантирующей их целостность — в данном случае то, что они не были подменены где-то между участвующими в связи сторонами.
- С помощью механизма описания *ролей*, определения доступности различных методов Web-компонентов и EJB-компонентов для разных ролей, а также задания политик переноса или создания ролей при совместной работе нескольких методов. Роли, политики их переноса и правила доступа различных ролей к методам описываются в дескрипторах развертывания компонентов.
  При развертывании приложения зарегистрированные на J2EE-сервере пользователи и группы пользователей могут быть отображены на различные роли, определенные в приложении.
- С помощью определения ограничений доступа к наборам ресурсов, задаваемых в виде списков унифицированных идентификаторов ресурсов (URI) или шаблонов URI. Эти ограничения описываются в дескрипторе развертывания приложения и определяют роли и разрешенные им виды прямого доступа (не через обращение к другим компонентам) к данному набору URI.
- С помощью программного определения ролей и пользователей, от имени которых работает текущий поток, из кода самих компонентов.
  Это можно делать при помощи методов `isUserInRole()` и `getUserPrincipal()` интерфейса `HTTPServletRequest`, используемого для представления запросов к Web-компонентам, и аналогичных методов `isCallerInRole()` и `getCallerPrincipal()` интерфейса `EJBContext`, используемого для описания контекста выполнения методов EJB-компонентов.

Работа с XML

Поскольку сейчас очень часто информация хранится и передается в виде XML-документов, для разработки Web-приложений большое значение имеют средства работы с XML. Основными элементами, необходимыми для облегчения работы с XML-документами, являются их разбор и внутреннее представление XML-данных.

Библиотеки для работы с XML-документами находятся в пакетах `javax.xml` , `org.w3c.dom` и `org.xml.sax`, а также вложенных в них пакетах. В этих пакетах определяются следующие интерфейсы.

- Общий интерфейс обработчиков XML находится в пакете `javax.xml.parsers`. Такие обработчики могут быть основаны на *простом интерфейсе работы с XML (Simple API for XML, SAX)* [15] или на *объектной модели документов (Document Object Model, DOM)* [16]. Оба этих подхода основаны на стандартах W3C.

- Простой интерфейс для работы с XML (SAX) [15] определяется в пакете `org.xml.sax`. Это интерфейс, основанный на событиях — XML-парсер, реализующий его, последовательно разбирает XML-данные и генерирует очередное событие в зависимости от вида обнаруженной конструкции. Для работы с XML-документами необходимо написать ряд обработчиков таких событий, являющихся реализациями интерфейса `org.xml.sax.ContentHandler`.
- Объектная модель документов (DOM) [16] представляет собой интерфейс работы с XML-документом, представленным в виде дерева его элементов. Java-представление этого интерфейса описано в пакете `org.w3c.dom`. Одной из его реализаций является JDOM [17], а dom4j [18] предоставляет несколько упрощенную и более удобную с точки зрения Java, но не вполне соответствующую стандарту DOM реализацию.
- Обработка XML-документов может быть построена на базе *расширяемого языка трансформаций на основе таблиц стилей (Extensible Stylesheet Language Transformations, XLST)* [19]. При этом процесс обработки документов описывается в виде XSLT-программы, которая затем подается на вход интерпретатору XSLT (XSLT-процессору) вместе с обрабатываемым XML-документом. Интерфейсы для работы с XSLT определены в пакете `javax.xml.transform`. Широко используемыми XSLT-процессорами являются Saxon [20] и Xalan [21].
- В новую версию J2EE 5.0, ожидаемую в 2006 году, должны войти интерфейсы для *потоковой обработки XML-документов (Streaming API for XML, StAX)*. В этом подходе XML-документ рассматривается как поток различных конструкций, которые становятся доступными по запросу (pull-модель), в отличие от работы на основе событий в SAX, когда каждая конструкция порождает событие, которое нужно обработать (push-модель). Обработка XML-документов в стиле StAX гораздо гибче SAX и вполне сравнима с ней по удобству. В настоящее время уже доступны спецификации интерфейсов StAX [22] и несколько их реализаций.

Платформа .NET

Среда .NET предназначена для более широкого использования, чем платформа J2EE. Однако ее функциональность в части, предназначенной для разработки распределенных Web-приложений, очень похожа на J2EE.

В рамках .NET имеются аналоги основных видов компонентов J2EE. Web-компонентам соответствуют компоненты, построенные по технологии ASP.NET, а компонентам EJB, связывающим приложение с СУБД, — компоненты ADO.NET. Компонентная среда .NET обычно рассматриваются как однородная. Однако имеющиеся небольшие отличия в правилах, управляющих созданием и работой компонентов ASP.NET и остальных, позволяют предположить, что в рамках .NET существует аналог Web-контейнера, отдельная компонентная среда для ASP.NET, и отдельная — для остальных видов компонентов. Тем не менее, даже если это так, эти среды тесно связаны и, как и контейнеры J2EE, позволяют взаимодействовать компонентам, размещенным в разных средах. Компонентная среда для ASP.NET, в отличие от Web-контейнера в J2EE, поддерживает автоматические распределенные транзакции.

В связи с этим, типовая архитектура Web-приложений на основе .NET может быть изображена так, как это сделано на Рис. 2.

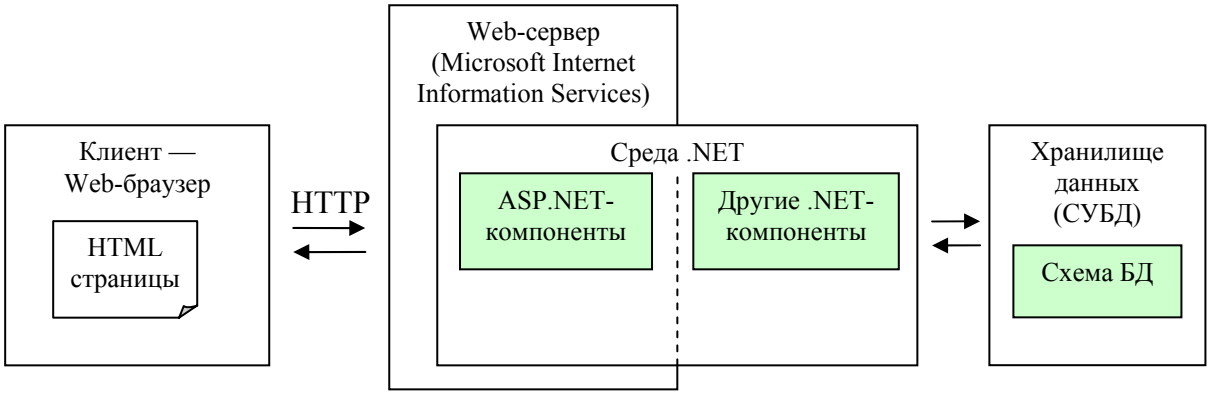

Рисунок 2. Типовая архитектура Web-приложения на основе .NET.

В целом, Web-приложения на основе .NET используют тот же набор архитектурных стилей, что и аналогичные J2EE-приложения.

Обычно .NET-компоненты представляют собой набор .NET-классов и *конфигурационных файлов*, играющих роль дескрипторов развертывания и так же представленных в некотором формате на основе XML. Для приложений в целом тоже пишутся особые конфигурационные файлы.

Связь

Связь между компонентами в рамках .NET осуществляется при помощи механизма Remoting, реализующего как RMI, так и асинхронную передачу сообщений.

Классы и интерфейсы, служащие основой механизма Remoting, находятся в пространстве имен `System.Rutime.Remoting` и его подпространствах, в сборках mscorlib и System.Rutime.Remoting.

В рамках Remoting объекты, которые могут участвовать в качестве целей удаленного вызова, его параметров и результатов, делятся на ***передаваемые по значению (marshal-by-value)*** и ***передаваемые по ссылке (marshal-by-reference)***.

Тип объекта, передаваемого по значению, должен реализовывать интерфейс `System.Runtime.Serialization.ISerializable` или должен быть помечен атрибутом `System.SerializableAttribute`. В последнем случае .NET автоматически создает код, необходимый для сериализации или десериализации данных объекта. Исключения, которые могут быть созданы методом, вызываемым удаленно, также должны быть передаваемы по значению.

Объекты, передаваемые по ссылке, должны наследовать классу `System.MarshalByRefObject`. При передаче такого объекта как аргумента или результата в другом процессе (или зоне приложения, см. следующий раздел) создается клиентская заглушка, связанная с этим объектом, и называемая в .NET ***посредником (proxy)***.

Более тонкую настройку удаленных вызовов можно делать при помощи наследников класса `System.ContextBoundObject`. Наследование этого класса говорит о том, что удаленные вызовы в объекте этого класса должны происходить в рамках некоторого контекста, являющегося частью его зоны приложения. Примером такого контекста служит контекст транзакции. При вызовах таких объектов .NET проводит дополнительные проверки, связанные с разницей контекстов между вызывающим объектом и вызывающим.

Ниже находится пример взаимодействующих по механизму Remoting классов, имеющий ту же функциональность, что и пример, приведенный для Java RMI.

Код класса, реализующего метод для удаленных обращений.

```
using System;

namespace Examples.Remoting
{
  public class Hello : MarshalByRefObject
  {
    public String HelloMethod() { return "Hello!"; }
  }
}
```

Код сервера, ожидающего обращения клиентов.

```
using System;
using System.Runtime.Remoting;

namespace Examples.Remoting
{
  public class HelloImpl
  {
    public static void Main()
    {
      RemotingConfiguration.Configure("RemotingServer.exe.config");
      Console.ReadLine();
    }
  }
}
```

Код клиентского класса.

```
using System;
using System.Runtime.Remoting;

namespace Examples.Remoting
{
  public class HelloClient
  {
    public static void Main()
    {
      RemotingConfiguration.Configure("RemotingClient.exe.config");
      Hello stub = new Hello();
      Console.WriteLine("response: " + stub.HelloMethod());
    }
  }
}
```

Кроме классов должны быть написаны конфигурационные файлы серверного и клиентского приложений. Конфигурационный файл сервера выглядит так.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown
          mode = "Singleton"
          type = "Examples.Remoting.Hello, Hello"
          objectUri = "MessageServer"
        />
      </service>
      <channels>
        <channel ref = "http" port = "8989" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Конфигурационный файл клиента.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type = "Examples.Remoting.Hello, Hello"
          url = "http://hostname:8989/MessageServer"
        />
        </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Чтобы запустить этот пример, нужно выполнить следующие шаги.

- Скомпилировать все классы (предполагается, что классы находятся в файлах с именами Hello.cs, Server.cs и Client.cs).
  ```
  csc.exe /noconfig /t:library Hello.cs
  csc.exe /noconfig /r:Hello.dll Server.cs
  csc.exe /noconfig /r:Hello.dll Client.cs
  ```
  В результате первой команды будет получена динамическая библиотека, содержащая определение класса `Hello`. В двух других она используется, чтобы предоставить ссылку на этот тип.
- Запустить серверный компонент
  ```
  Server.exe
  ```
- Перенести на другую машину или в отдельную директорию на той же машине файлы Client.exe, Hello.dll, RemotingClient.exe.config, заменив `hostname` в URL в конфигурационном файле клиента на имя машины, где работает сервер, и запустить клиентский компонент
  ```
  Client.exe
  ```
  Если никаких ошибок не сделано, и порт 8989 на серверной машине доступен, клиент выдаст сообщение.
  ```
  response: Hello!
  ```

С помощью конфигурационных файлов можно определить политику активации серверного объекта (использовать один объект для всех вызовов, создавать для каждого вызова свой объект, создавать объект при его создании клиентом), транспортный протокол для передачи сообщений (HTTP или TCP), и пр.

Асинхронное взаимодействие в рамках .NET может быть реализовано с помощью имеющихся в библиотеке .NET асинхронных делегатов, на основе асинхронных удаленных вызовов по Remoting или с помощью обмена сообщениями на базе библиотек `System.Messaging`.

Взаимодействие между компонентами ASP.NET использует похожие, но скрытые от разработчика компонентов механизмы.

Именование

В примере на взаимодействие по Remoting использовалась локальная служба именования, встроенная в среду .NET — поэтому никаких специальных действий по регистрации и поиску компонентов не производилось. Эта служба позволяет не заботиться об этих вопросах в том случае, если физическое положение компонентов, с которыми необходимо установить связь, известно и постоянно.

Если же это не так, необходимо иметь полноценную службу именования и/или службу каталогов. В этом качестве в .NET используется Active Directory. Поскольку эта технология появилась раньше, чем среда .NET, в рамках .NET была создана библиотека адаптеров, позволяющих использовать функции Active Directory. Элементы этой

библиотеки находятся в пространстве имен `System.DirectoryServices` и его подпространствах, в сборках System.DirectoryServices и System.DirectoryServices.Protocols.

Для работы с записями Active Directory используются объекты класса `System.DirectoryServices.DirectoryEntry`. С помощью конструкторов и методов этого класса можно создавать и изменять регистрационные записи службы каталогов. Для поиска зарегистрированных объектов по идентификаторам или свойствам используются методы класса `System.DirectoryServices.DirectorySearcher`.

Active Directory поддерживает обращения к своим записям по нескольким разным протокола, включая протокол LDAP.

Процессы и синхронизация

Помимо процессов и потоков, среда .NET поддерживает так называемые *зоны приложений (application domains)*, которые служат агрегатами ресурсов, как и процессы, но, в отличие от них, управляются с помощью более эффективных механизмов. В рамках одного процесса может быть создано несколько зон приложений. Передача объектов и ресурсов между зонами приложений невозможна без использования специальных механизмов, таких как Remoting. Потоки же в .NET могут пересекать границы зон приложений, если обладают соответствующими правами.

Зоны приложений служат дополнительным элементом защиты .NET-приложений от непреднамеренного взаимного влияния и позволяют сохранить работоспособность процесса при возникновении проблем в одном из его приложений.

Помимо автоматически создаваемых потоков и зон приложений, разработчик может создавать свои собственные потоки и зоны приложений. Вопросы синхронизации потоков и передачи данных между зонами приложений в Web-приложениях могут решаться при помощи стандартных механизмов .NET — конструкций и библиотек синхронизационных примитивов, а также библиотечного класса `System.AppDomain`, чьи методы позволяют выполнять различные операции с зонами приложений.

Целостность

Целостность данных в .NET поддерживается, как и в J2EE, в основном, за счет механизма транзакций. Распределенные транзакции в .NET реализованы на базе *сервера транзакций Microsoft (Microsoft Transaction Server, MTS)*, который появился как часть компонентной среды COM+. Интерфейсы для работы с его функциями собраны в пространстве имен `System.EnterpriseServices`, в сборке с таким же именем.

Автоматические транзакции поддерживаются при помощи указания у классов транзакционных атрибутов, имеющих тип `System.EnterpriseServices.TransactionAttribute`. Такой атрибут может принимать значения Required, RequiresNew, NotSupported, Supported и Disabled. Первые три имеют то же значение, что и в J2EE. Атрибут Supported действует аналогично Supports в J2EE. Атрибут Disabled обозначает, что транзакционный контекст вызвавшего метода будет проигнорирован.

Значение атрибутов по умолчанию для методов компонентов ASP.NET и методов обычных классов, используемых в распределенных транзакциях, различны — у первых это Disabled, у вторых — Required.

Чтобы определить класс, чьи методы могут участвовать в транзакциях, нужно унаследовать его от класса `System.EnterpriseServices.ServicedComponent`, определить транзакционный атрибут, а также прикрепить к сборке, содержащей этот класс, сертификат и зарегистрировать эту сборку в реестре COM+.

Для завершения или отката автоматической транзакции используются следующие конструкции.

- Атрибут `System.EnterpriseServices.AutoCompleteAttribute` у участвующего в транзакции метода говорит о том, что в случае нормального завершения работы метода и отсутствия проблем у других ее участников транзакция будет завершена успешно. Если же в результате работы метода будет создано исключение, транзакция будет отменена.
- Методы `SetComplete()` и `SetAbort()` класса `System.EnterpriseServices.ContextUtil` могут использоваться для успешного завершения или отката автоматически созданной транзакции.

Создание и управление транзакциями «в ручном режиме» может быть осуществлено для компонентов ADO.NET при помощи методов класса `System.Data.Common.DbConnection` и его наследников и класса `System.Data.Common.DbTransaction`. Для управления транзакциями при передаче сообщений используется класс `System.Messaging.MessageQueueTransaction`.

Отказоустойчивость

Так же, как и для J2EE, отказоустойчивость .NET-приложений должна обеспечиваться либо за счет использования дополнительных продуктов, либо за счет специфического проектирования приложения.

Защита

Защищенность .NET-приложений поддерживается примерно теми же методами, что и защищенность J2EE-приложений.

Здесь также имеется несколько техник аутентификации, возможность определения ролей, обеспеченных набором прав доступа к различным элементам системы, а также возможность использования различных протоколов шифрования и защищенной передачи данных, управления ключами и подтверждения целостности данных. В рамках .NET используются также многоуровневые *политики защиты*, определяющие набор прав, предоставляемых коду из разных источников.

Программные интерфейсы к различным механизмам управления защищенностью приложений и ресурсов реализуются классами и интерфейсами пространства имен `System.Security`, находящимися в сборках mscorlib, System и System.Security.

Работа с XML

В целом техника работы с XML-документами в .NET опирается на реализацию объектной модели документов XML (DOM) и на механизм разбора, аналогичный StAX, реализуемый классом `System.Xml.XmlReader`. Классы, реализующие различные парсеры XML, различные варианты представления XML-документов, а также их трансформацию на основе XSLT-описаний, находятся в пространстве имен `System.Xml`, разбросанному по сборкам System.Data, System.Data.SqlXml и System.Xml.

Одной из особенностей работы с XML в .NET является встроенная возможность работы с XML-данными в рамках механизмов ADO.NET (в основном предназначенных для работы с реляционными СУБД) с помощью класса `System.Xml.XmlDataDocument`.

Литература к Лекции 13

[1] Web-сайт консорциума World Wide Web http://www.w3.org/.
[2] http://www.xml.com/.
[3] XML 1.1, 2004. Доступен через http://www.w3.org/TR/xml11/.
[4] Annotated XML 1.0, 1998. Доступен через http://www.xml.com/axml/axml.html.
[5] Расширяемый язык разметки (XML) 1.0 (русский перевод первой версии стандарта). Доступен через http://www.rol.ru/news/it/helpdesk/xml01.htm.

[6] Материалы по XMLSchema http://www.w3.org/XML/Schema.
[7] Namespaces in XML, 1999. Доступен через http://www.w3.org/TR/REC-xml-names/.
[8] Java Platform Enterprise Edition Specifications, version 1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.
[9] Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003.
[10] Документация по Java RMI http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html
[11] Документация по JMS API http://java.sun.com/products/jms/docs.html
[12] Документация по JNDI http://java.sun.com/j2se/1.5.0/docs/guide/jndi/index.html
[13] RFC 1766, доступен по ссылке http://rfc.net/rfc1766.html
[14] http://java.sun.com/products/jndi/serviceproviders.html
[15] Web-страница стандарта SAX http://www.saxproject.org/
[16] Web-страница стандарта DOM http://www.w3.org/DOM/
[17] Web-страница проекта JDOM http://www.jdom.org/
[18] Web-страница проекта dom4j http://www.dom4j.org/
[19] Стандарт XSLT. Доступен через http://www.w3.org/TR/xslt
[20] Web-страница проекта Saxon http://www.saxonica.com/
[21] Web-страница проекта Xalan http://xml.apache.org/xalan-j/
[22] Streaming API for XML http://www.jcp.org/en/jsr/detail?id=173
[23] B. McLaughlin. Java and XML, Second Edition. O'Reilly, 2001.
[24] Документация по платформе J2EE http://java.sun.com/j2ee/1.4/docs/index.html
[25] Документация по платформе .NET — находится в разделе .NET Development MSDN, http://msdn.microsoft.com/library/default.asp
[26] П. Аллен, Дж. Бамбара, М. Ашнаульт, Зияд Дин, Т. Гарбен, Ш. Смит. J2EE. Разработка бизнес-приложений. СПб.: ДиаСофт, 2002.

Задания к Лекции 13