Tomcat 4

Версия 1.01 от 15.09.2002

Содержание

Java и Web

Технологии Servlets и JSP

Сервлеты – это особым образом написанные (согласно спецификации) Java-программы, которые выполняются удаленно на сервере и вызов которых осуществляется удаленно из web-броузера с помощью HTTP протокола через web-сервер. Сервлеты могут выполнять все те же функций,

которые выполняются CGI-скриптами, только вместо Perl, Pytnon или C++ в данном случае используется язык Java, что дает много преимуществ по сравнению с CGI. Эти преимущества заключаются в удобстве написания, поддержки и изменения кода, а также заключаются в самом способе исполнения Java-программ на сервере.

Полное описание технологии Servlets можно найти по данному адресу: http://java.sun.com/products/servlet/. Текущая версия спецификации – 2.3.

Вскоре после появления технологии сервлетов web-разработчикам все-таки пришлось столкнуться с одним большим неудобством: для генерации HTML-страниц на лету с помощью сервлета весь HTML-код приходилось помещать в сами сервлеты. Получалось, что HTML-код страницы (presentation) смешивался с Java-кодом (logic), что затрудняло работу как программиста так и дизайнера сайта.

Для решения этой проблемы была придумана технология JSP - JavaServer Pages. Она во многом напоминала существовавшие тогда технологии ASP (от Microsoft) и ColdFusion (от Allaire). Но сходство было чисто внешним. Так же как и в ASP и в ColdFusion в JSP вы могли вставлять Java-код прямо в HTML-код страницы, но если в ASP и ColdFusion этот код при каждом вызове страницы интерпретировался, в JSP при первом вызове jsp-страницы этот код незримо от разработчика переделывался в сервлет и компилировался, после чего при последующих обращениях веб-сервер вызывает уже не саму jsp-страницу, а откомпилированный сервлет. Разумеется при внесении изменений в jsp-страницу веб-сервер обнаруживает, что страница изменилась и снова обновляет сервлет, соответствующий этой странице.

Полное описание технологии JavaServer Pages можно найти по данному адресу: http://java.sun.com/products/jsp/. Текущая версия спецификации – 1.2.

В технологиях Servlets и JSP введено понятие "контейнера" (container). Если вкратце, то Servlets-контейнер – это движок, отвечающий за выполнение Servlet-ов. JSP-контейнер – это движок, отвечающий за преобразование js-страниц в сервлеты и передачу этих сервлетов Servlets-контейнеру. Так как сервлеты и jsp-страницы вызываются через HTTP-протокол, то Servlets-контейнер и JSP-контейнер часто сопровождает еще один компонент – web-сервер, который тоже может быть написан на Java.

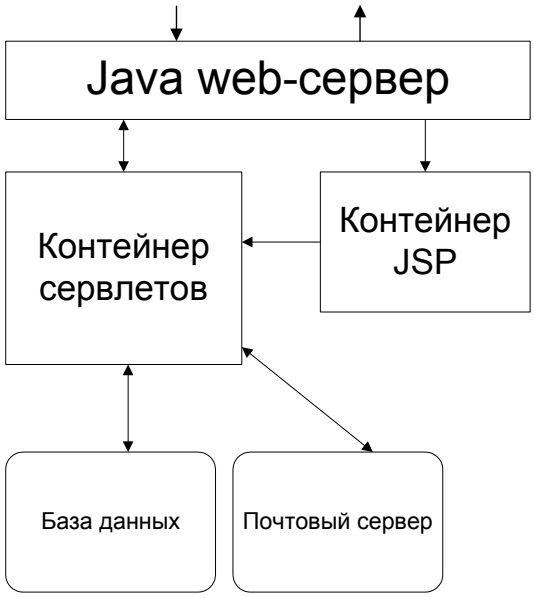


Рис. 1 Java веб-сервер и контейнеры

Как видно на Рис. 1, веб-сервер, написанный на Java, получает запросы от броузера на выполнение того или иного сервлета или jsp-страницы на сервере, передает запрос в контейнер, который выполняет тот или иной сервлет. Результаты выполнения возвращаются контейнером веб-серверу, который в свою очередь пересылает его броузеру.

В качестве веб-сервера может служить и обычный веб-сервер типа Microsoft IIS или Apache. На Рис. 2 представлен пример работы Servlet- и JSP-контейнеров с веб-сервером Apache. Apache (или IIS) настраиваются таким образом, что запросы к обычным HTML-файлам и CGI-скриптам обрабатываются, как обычно, а запросы к сервлетам и jsp-страницам перенаправляются в Servelts-контейнер.

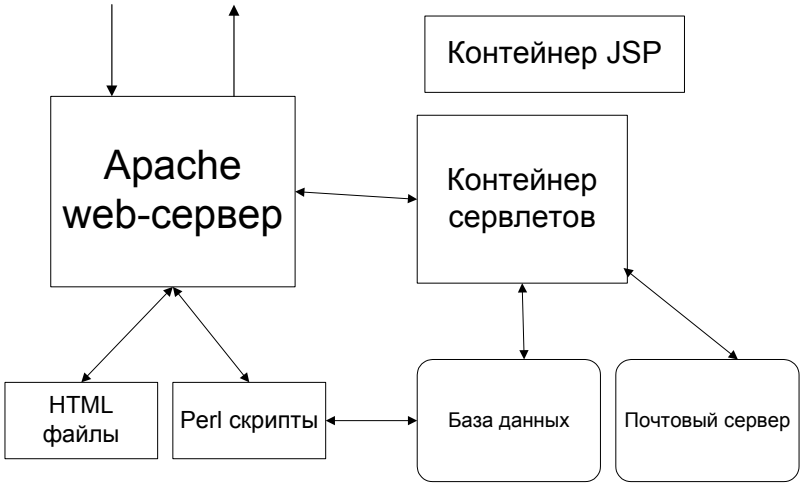


Рис. 2 Стандартный веб-сервер и контейнеры

J2EE и сервера приложений

Технологии Servlets и JSP были объединены с несколькими другими Java-технологиями, и этот комплект технологий был назван Java 2 Enterprise Edition.

Так появились сервера Java-приложений от различных компаний (IBM, BEA, IONA, Borland), в том числе и от самой Sun. Каждая компания в своем сервере приложений внутренне реализует технологии J2EE по своему, но внешне все они соответствуют спецификациям, написанным Sun. Если сервер приложений следует этим спецификациям в точности, и Sun это подтверждает, то сервер приложений получает право называться "J2EE-compliant Application Server" ("соответствующий спецификациям J2EE") или кратко – "J2EE Application Server". Разумеется, J2EE – это минимум, что должен уметь J2EE AS, как правило компании снабжают свои сервера еще некоторыми дополнительными функциями, компонентами и технологиями, которые не охвачены спецификацией J2EE, что делается для повышения привлекательности своего продукта.

Раз, как уже было сказано выше, в спецификацию J2EE включены спецификация Servlets и JSP, то каждый J2EE AS обязательно умеет работать как web-сервер и умеет выполнять сервлеты и jsp-страницы.

Описание всех технологий, входящих в комплект J2EE, можно найти по адресу: http://java.sun.com/j2ee/.

Sun взяла на себя обязанность выпускать так называемые "reference implementations" ("образцовые реализации") своих спецификаций. Эти программные продукты являются как бы эталонным исполнением в коде того, что написано на бумаге. Например, Sun позволяет загрузить со своего сайта бесплатную эталонную реализацию сервера J2EE-приложений

(http://java.sun.com/j2ee/download.html), по которой (как по минимальной планке) должны равняться все остальные компании.

Разумеется эта бесплатная версия не предназначена для использования в реальной среде, а только для обучения, разработки и тестирования J2EE-компонентов. Для реальной среды Sun выпускает платную коммерческую версию сервера J2EE-приложений (http://wwws.sun.com/software/products/appsrvr/).

Наконец, Sun когда-то раньше предлагала бесплатную эталонную реализацию Java веб-сервера с Servlet- и JSP-контейнерами, которая называлась JServ. После выхода в свет технологии J2EE весь код был передан Apache Software Foundation, а продукт поменял свое название на Tomcat. Впрочем, сам Tomcat для Sun не перестал существовать. Он включен в состав эталонной реализации J2EE-сервера, с тем только, что теперь разработка Tomcat-а перепоручена обществу вольных программистов.

Apache и Jakarta Tomcat

Apache Software Foundation (http://www.apache.org/) вам знакомо наверняка по самому главному продукту – веб-серверу Apache. Но этот продукт – не единственный. Все множество проектов Apache сгруппировано в под-проекты. Так, например, все проекты связанные с реализацией серверных технологий на Java, попадают в группу проектов под названием "Jakarta".

Проект Jakarta (http://jakarta.apache.org/) – это одна и групп Apache Software Foundation, которая занимается созданием и поддержкой качественных бесплатных серверных приложений с открытым исходным кодом и построенных на Java-платформе. Разумеется, когда Sun передала Apache бразды правления, проект Tomcat попал в группу проектов Jakarta.

Итак, Tomcat (http://jakarta.apache.org/tomcat/) – является эталонной реализацией веб-сервера, Servlets- и JSP-контейнеров. Tomcat разрабатывается открытым обществом программистов и распространяется под лицензией Apache Software License (http://www.apache.org/licenses).

К настоящему времени выпущено несколько версий Tomcat, которые эталонно реализуют следующие версии спецификаций:

Спецификация Servlet/JSP	**Версия Tomcat**
2.2 / 1.1	3.3.1
2.3 / 1.2	4.1.10
2.4 / 2.0	5.0.x (пока не выпущена)

Версия Tomcat 3.3.1 является последней стабильной версией, реализующей спецификаций Servlets 2.2 / JSP 1.1. "Последней" значит, что работа над этой версией завершена и в нее вносятся лишь небольшие дополнения и исправления ошибок. До сих пор во многих компаниях работают сервера J2EE-приложений, которые реализуют эти старые версии спецификаций. Если вы планируете разрабатывать приложения для старых версий серверов, воспользуйтесь Tomcat 3.3.1.

Tomcat четвертого поколения (4.x) не только поддерживает новые версии спецификаций, изменилась и его внутренняя архитектура. Был с нуля написан новый контейнер сервлетов (этот движок называется Catalina), повышена скорость работы и уменьшены требования к памяти.

В 2001 году вышла стабильная версия Tomcat 4.0.1 (), которая полностью реализовывала спецификации Servlets 2.3 / JSP 1.2, и (опять-таки согласно спецификации) позволяла выполнять код, написанный по старым спецификациям (обратная совместимость). В настоящее время для

загрузки доступна версия Tomcat 4.0.4, которая является все тем же Tomcat 4.0.1 только исправленным и улучшенным.

Страница Tomcat 4.0.x: http://jakarta.apache.org/tomcat/tomcat-4.0-doc/index.html

Загрузка: http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.0.4/

Между тем сообщество программистов не остановилось на достигнутом. В очередной раз было принято решение переписать Tomcat по новой, в результате чего появилась версия Tomcat 4.1.x, в которой помимо всего того, что сделано в Tomcat 4.0.x, появился ряд улучшений и значительных дополнений, а именно:

- JMX based administration features
- JSP and Struts based administration web application
- New Coyote connector (HTTP/1.1, AJP 1.3 and JNI support)
- Rewritten Jasper JSP page compiler
- Performance and memory efficiency improvements
- Enhanced manager application support for integration with development tools
- Custom Ant tasks to interact with the manager application directly from build.xml scripts

Стабильной версией Tomcat ветки 4.1.x на момент написания книги является Tomcat 4.1.10.

Страница Tomcat 4.1.x: http://jakarta.apache.org/tomcat/tomcat-4.1-doc/index.html

Загрузка: http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.1.10/

Данная книга освещает Tomcat 4.0.x. По мере пополнения материала мы уделим место и новшествам Tomcat 4.1.x.

Tomcat пятого поколения (Tomcat 5.x) будет реализовывать спецификации Servlet 2.4 и JSP 2.0, которые в настоящее время находятся еще только на завершающей стадии обсуждения.

На заметку. Tomcat не единственный веб-сервер и jsp/servlet-контейнер. Resine и Jigsaw.

Установка Tomcat

Архитектура Tomcat

Web-приложение

http://www.onjava.com/lpt/a/671

http://www.onjava.com/lpt/a/780

Основы настройки

Tomcat 4.0 не использует переменную среды `CLASSPATH` для поиска классов, которые в ней объявлены. Поэтому, если у вас возникнет вопрос, почему Tomcat или приложения Tomcat-а жалуются на то, что невозможно найти какой-то класс, который вы занесли в переменную `CLASSPATH`, вспомните, что все классы, которые там объявлены, Tomcat-ом игнорируются. Впрочем, вы можете изменить это поведение, поменяв соответствующим образом файл запуска Tomcat-а – `startup.bat`/`startup.sh`. Но лучше всего поместить нужные классы и jar-библиотеки в один из следующих каталогов:

/WEB-INF/classes/*.class
/WEB-INF/lib/*.jar
<JAVA_RUNTIME>/lib/ext

tomcat/common/classes/*.class
tomcat/common/lib/*.jar
tomcat/classes/*.class
tomcat/lib/*.jar

Если jar-библиотека или класс предназначены только для Tomcat-а и приложений, в нем работающих, вы можете поместить их в каталог `<TOMCAT_HOME>/common/classes` или `<TOMCAT_HOME>/common/lib`.

Подробности о server.xml

Конфигурация Tomcat осуществляется с помощью файла `server.xml`, в котором содержится вся информация о настройке, записанная в XML-формате. Этот XML-файл – сердце Tomcat.

По умолчанию, когда вы запускаете Tomcat с помощью команды `startup.bat` или `startup.sh`, используется server.xml, который расположен в каталоге `<TOMCAT_HOME>/conf/`. Все изменения, сделанные в этом файле, отражаются на том, как и какие компоненты Tomcat будет использовать в своей работе. Я не советую менять этот файл, так как там содержится множество полезных комментариев и вариантов конфигураций, которые могут вам пригодиться в качестве справочника. Вместо этого, вы можете создать свой `server.xml`, и подсунуть его Tomcat-у следующим образом:

```
<TOMCAT_HOME>/bin/startup.bat -config my_server.xml
```

В `my_server.xml` вы можете определить только те компоненты и настройки, которые необходимы для вашего конкретного проекта. Вышеприведенную строку вы можете поместить в `.bat` или `.sh` файл или в Windows создать ярлык (Shortcut), после чего сможете запускать Tomcat в различной конфигурации и нужный вам проект одним щелчком мышки.

Например, я работаю над четырьмя веб-проектами одновременно. Каждый проект хранится в своем каталоге. В каждом из каталогов хранится и свой `server.xml`, который инициализирует

компоненты нужные мне только для данного проекта. На "Рабочем Столе" у меня четыре ярлыка (Shortcut), в каждом из которых прописана вышеуказанная команда с соответствующим файлом `server.xml`.

Давайте рассмотрим основные составные части этого файла. Для этого вам необходимо открыть файл `server.xml` в обычном текстовом редакторе. В установке Tomcat по умолчанию `server.xml` располагается в каталоге `<TOMCAT_HOME>/conf/`.

Упрощенная структура `server.xml` выглядит так:

```
<Server>
  <Service>
    <Connector />
    <Engine>
      <Host>
        <Context />
      </Host>
    </Engine>
  </Service>
</Server>
```

Ниже представлен вариант `server.xml`, в котором удалены комментарии и некоторые компоненты, о которых разговор пойдет позднее.

Пример 1. Простой файл server.xml

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Service name="Tomcat-Standalone">
    <Connector className="org.apache.catalina.connector.http.HttpConnector"
      port="8080" minProcessors="5" maxProcessors="75"
      enableLookups="true" redirectPort="8443"
      acceptCount="10" debug="0" connectionTimeout="60000"/>
    <Engine name="Standalone" defaultHost="localhost" debug="0">
      <Logger className="org.apache.catalina.logger.FileLogger"
        prefix="catalina_log." suffix=".txt"
        timestamp="true"/>
      <Realm className="org.apache.catalina.realm.MemoryRealm" />
      <Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
        <Valve className="org.apache.catalina.valves.AccessLogValve"
          directory="logs" prefix="localhost_access_log." suffix=".txt"
          pattern="common"/>
        <Logger className="org.apache.catalina.logger.FileLogger"
         directory="logs" prefix="localhost_log." suffix=".txt"
         timestamp="true"/>
        <Context path="/examples" docBase="examples" debug="0"
         reloadable="true">
          <Logger className="org.apache.catalina.logger.FileLogger"
           prefix="localhost_examples_log." suffix=".txt"
           timestamp="true"/>
         </Context>
      </Host>
    </Engine>
  </Service>
</Server>
```

Элемент <Server>

Первый элемент, который встречается нам в файле `server.xml` – это элемент <Server>. Этот элемент представляет весь контейнер. Он является корневым элементом в XML-файле.

Элементу <Server> соответствует интерфейс `org.apache.catalina.Server`. Интерфейс Server является singleton-ом, и он представляет собой экземпляр Tomcat в памяти. В элементе <Server> могут располагаться один или несколько элементов <Service>. Ниже мы перечислим, какие атрибуты могут быть использованы в элементе <Server>.

className: Полное название класса, который реализует интерфейс `org.apache.cataline.Server`. Если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardServer`.

port: Номер TCP/IP порта, от которого Tomcat ожидает команду об остановке сервера. Команда должна быть отправлена клиентом, работающим на той же машине, где работает Tomcat. Этот атрибут обязателен.

shutdown: Этот атрибут определяет саму команду, по которой Tomcat прекратит свою работу. Эта команда должна быть отправлена на порт, указанный в предыдущем атрибуте. Этот атрибут обязателен.

Элемент <Server> в примере, что мы привели выше, выглядит так:

```
<Server port="8005"
  shutdown="SHUTDOWN"
  debug="0">
```

> **Внимание**. Атрибут **debug** может быть использован во всех элементах XML-файла. Он определяет уровень отладки (debug level), согласно которому компоненты Logger будут писать в логи отладочные сообщения. От компонентах Logger мы поговорим чуть позднее.

Элемент <Server> не может располагаться внутри какого-либо другого элемента. Он является родительским элементом для элемента <Service>.

Элемент <Service>

Следующим в иерархии элементов в файле server.xml идет элемент <Service>. Этот элемент представляет интерфейс `org.apache.catalina.Service`. Ниже приводится список допустимых атрибутов данного элемента.

className: Полное название класса, который реализует интерфейс `org.apache.cataline.Service`. Если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardService`.

name: Определяет имя сервиса. Это имя будет упомянуто в логах. Имя каждого сервиса, расположенного в одном и том же элементе <Server>, должно быть уникальным.

Элемент <Service> в примере, что мы привели выше, описывает режим работы Tomcat-а в качестве самостоятельного веб-сервера. Код выглядит следующим образом.

```
<Service name="Tomcat-Standalone">
```

О том, как добавлять другие варианты элемента `<Service>`, мы поговорим в следующей главе.

Элемент <Service> должен быть вложен в элемент <Server>. Сам он может включать элементы <Connector> и <Engine>.

Элемент <Engine>

Третьим элементом в иерархии XML-файла server.xml является элемент <Engine>. Он представляет контейнер сервлетов Catalina. В каждом элементе <Service> может быть только один элемент <Engine>. Этот единственный компонент <Engine> будет обрабатывать все запросы, поступающие к нему через компоненты <Connector>. Элемент <Engine> должен идти сразу же после последнего элемента <Connector>. Все элементы <Connector> должны идти до элемента <Engine>, после него элементы <Connector> помещать нельзя.

Элемент <Engine> соответствует интерфейсу `org.apache.catalina.Engine`. Ниже представлен список допустимых атрибутов элемента <Engine>.

className: Полное название класса, который реализует интерфейс `org.apache.cataline.Engine`. Если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardEngine`.

defaultHost: Если на вашей машине будет работать несколько виртуальных серверов данный атрибут указывает имя хоста, на который будут направляться по умолчанию все запросы, если в заголовке запроса не указано, к какому именно хосту этот запрос направлен. Имя каждого виртуального хоста задается атрибутом в элементе <Host>. Имя, указанное в атрибуте defaultHost обязательно должно совпадать с именем, указанным в одном из элементов <Host>.

name: Логическое имя контейнера. Оно будет использоваться при записи логов. Может быть любым. Атрибут обязателен.

Элемент <Engine> в примере, что мы привели выше, описывает контейнер с названием "Standalone". По умолчанию все запросы направляются хосту под именем "localhost". Код выглядит следующим образом.

```
<Engine name="Standalone" defaultHost="localhost" debug="0">
```

Элемент <Engine> должен находиться внутри элемента <Service>. Сам элемент <Engine> может включать в себя такие элементы, как

- <Logger>
- <Realm>
- <Valve>
- <Host>

Элемент <Host>

Элемент <Host> служит для описания виртуальных хостов в контейнере <Engine>. На каждом из хостов может работать одно или несколько веб-приложений, которые описываются элементов <Context> внутри каждого соответствующего элемента <Host>. Об элементе <Context> речь пойдет ниже.

В контейнере (в элементе <Engine>) должен быть описан как минимум один хост (элемент <Host>). Ниже представлен список допустимых атрибутов элемента <Host>:

className: Полное название класса, который реализует интерфейс `org.apache.catalina.Host`. Если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardHost`.

appBase: В этом атрибуте указывается путь к корневому каталогу, в котором будут располагаться все приложения, принадлежащие данному хосту. Путь может быть абсолютным и, значит, характерным для данной операционной системы, либо может быть относительным.

Относительный путь определяется относительно переменной <CATALINA_HOME>. Если данный атрибут не указан, по умолчанию используется относительный путь и каталог "webapps". Это значит, что хост будет искать свои приложения в каталоге <CATALINA_HOME>/webapps/.

> **На заметку**. Корневой каталог хоста вовсе не обязательно должен располагаться в том же каталоге, куда установлен Tomcat. Используя абсолютный путь, вы можете указать каталог, который располагается в любом месте на любом диске вашей машины.
>
> Я предпочитаю хранить все свои проекты в отдельном каталоге, так что при удалении старой версии Tomcat и установке новой, я не боюсь потерять какой-либо из своих проектов. Например, мой вариант атрибута **appBase** выглядит так:
>
> ```
> C:\web-projects\SuperBankProject\webapps\
> ```

unpackWARs: Этот атрибут указывает, должен ли хост распаковывать WAR-файлы, или нет. Если используется значение "false", хост запустит приложение, находящееся в WAR-файле, без его распаковки на диск. Если атрибут отсутствует, по умолчанию его значение равно "true".

name: Задает имя виртуального хоста. Этот атрибут обязателен. Имя не должно совпадать с именем другого хоста (элемент <Host>), находящегося в данном контейнере (элемент <Engine>).

Элемент <Host> в примере, что мы привели выше, выглядит следующим образом.

```
<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
```

Этот элемент <Host> описывает хост с именем "localhost". Для того, чтобы обратиться к нему, необходимо в броузере набрать в броузере следующий URL:

```
http://localhost:8080/
```

Так как в атрибуте **appBase** используется относительный путь, корневым каталогом данного хоста является каталог <CATALINA_HOME>/webapps/. Все WAR-файлы, которые данный хост обнаружит в данном каталоге при запуске, будут распакованы на диск потому что атрибуту **unpackWARs** присвоено значение "true".

Внимание: Номер порта, который добавляется к URL, задается элементом <Connector>, о котором речь пойдет ниже.

Элемент <Host> должен располагаться внутри элемента <Engine>. Сам он может включать в себя следующие элементы:

- <Logger>
- <Realm>
- <Valve>
- <Context>

Элемент <Context>

Элемент <Context> встречается в файле server.xml чаще всего. Он описывает определенное веб-приложение, которое будет запущено данным хостом (элемент <Host>). В элементе <Host> может присутствовать любое количество элементов <Context>. Единственное исключение – у каждого контекста-приложения (элемент <Context>) должен быть уникальным атрибут **path**.

На заметку. Если WAR-файлу присвоить имя `ROOT.war` и поместить его в корневой каталог хоста, этот WAR-файл станет корневым приложением данного хоста и будет доступен по данному URL-у:

```
http://<server>:<port>/
```

Все остальные контексты будут доступны по URL-у:

```
http://<server>:<port>/<имя контекста>/
```

Ниже представлен список допустимых атрибутов элемента <Context>.

className: Полное название класса, который реализует интерфейс `org.apache.catalina.Context`. Если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardContext`.

cookies: Определяет, хотите ли вы использовать cookie для отслеживания сессии, или нет. По умолчанию значение данного атрибута равно "true". Если оно равно "false", для отслеживания сессии вам придется пользоваться функцией URL rewrite.

crossContext: Если данному атрибуту присвоено значение "true", ваше приложение получает возможность с помощью метода `ServletContext.getContext()`получать доступ к другим контекстам-приложениям, запущенным на данном хосте. О умолчанию данному атрибуту присвоено значение "false". Это значит, что по умолчанию приложению запрещено "общаться" с другими приложениями.

docBase: Указывает, в каком каталоге находятся файлы веб-приложения. В данном атрибуте можно указать как относительный путь, так и абсолютный.

На заметку. Если воспользоваться абсолютным путем, то практически каждое приложение может находиться в любом каталоге на любом диске вашего компьютера. Таким образом вы получаете возможность разделить по каталогам на диске не только хосты, но даже приложения, которые будут запущены хостом.

path: Указывает имя контекста, под которым будет зарегистрировано данное приложение. Если в элементе <Host> используется несколько элементов <Context>, атрибут **path** данного контекста не должен совпадать с атрибутом **path** других контекстов.

Внимание. Давайте уясним разницу между атрибутами **docBase** и **path**. Ваше приложение может находиться в любом каталоге на любом диске, и его месторасположение указывается в атрибуте **docBase**. Атрибут **path** по сути определяет имя вашего приложения для данного хоста. Это имя используется в URL запроса. Например, если в атрибуту **path** присвоено значение "MyApp", то URL по которому данное приложение будет доступно, будет выглядеть следующим образом:

```
http://<server>:<port>/MyApp/
```

reloadable: Если данному атрибуту присвоено значение "*true*", Tomcat будет регулярно проверять подкаталоги `WEB-INF/classes/` и `WEB-INF/lib` данного контекста на предмет того, не появились ли там новые версии классов или JAR-библиотек. Значение "*true*" для данного атрибута имеет смысл присваивать только тогда, когда вы разрабатываете свое веб-приложение, так как в этом режиме скорость работы Tomcat-а сильно снижается.

wrapperClass: Определяет Java-класс, реализующий интерфейс `org.apache.catalina.Wrapper`. если имя класса не указано, по умолчанию используется класс `org.apache.catalina.core.StandardWrapper`.

useNaming: Если данному атрибут присвоено значение "*true*", контейнер сделает доступным для вашего приложения контекст JNDI. Иногда это необходимо для приложений, которые созданы по правилам J2EE. По умолчанию атрибут имеет значение "*true*".

override: Если данному атрибут присвоено значение "*true*", для данного контекста отменяются настройки, описанные в элементе <DefaultContext>.

> **На заметку**. <DefaultContext> – это элемент, расположенный в элементе <Host>, который задает настройку любого контекста, для которого отсуствует свой особый элемент <Context>. Если элемент <DefaultContext> присутствует в элементе <Engine>, его настройки и параметры действуют на все контексты всех хостов, что присутствуют в элементе <Host>.

По умолчанию значение атрибута равно "*false*".

workDir: Указывает, какой каталог должен использовать Tomcat для хранения временных файлов и откомпилированных классов для данного контекста. Для того, чтобы узнать из веб-приложения путь к этому каталогу, вызовите атрибут контекста под именем `java.servlet.context.tempdir`. Если данному атрибуту не присвоено никакое значение, Tomcat будет хранить все файлы в подкаталоге `<CATALINA_HOME>/work`.

Элемент <Context> в примере, что мы привели выше, выглядит следующим образом:

```
<Context path="/examples" docBase="examples" debug="0" reloadable="true">
```

Он описывает контекст под именем "examples". Все ресурсы данного приложения-контекста "examples" располагаются в каталоге `<TOMCAT_HOME>/webapps/examples`. В описании указано, что Tomcat должен проверять, не изменились ли классы данного контекста, и если изменились, он должен их перезагрузить.

Элемент <Context> должен располагаться в элементе <Host>. В самом же элементе <Context> могут располагаться следующие элементы:

- <Logger>
- <Loader>
- <Realm>
- <Manager>
- <Ejb>
- <Environment>
- <Parameter>
- <Resource>
- <ResourceParams>

> **Внимание**: Если вы не опишите контекст с помощью элемента <Context>, он будет загружен с настройками по умолчанию, которые описаны в файле web.xml, расположенном в каталоге `<CATALINA_HOME>/conf/`.

Теперь, несколько кратких правил, по которым взаимодействуют элементы <Host> и <Context>.

В элементе <Host> могут вообще отсутствовать элементы <Context>. В этом случае каждый подкаталог в корневом каталоге данного хоста будет считаться приложением-контекстом.

Если в <Host> присутствуют элементы <Context>, но в корневом каталоге есть и другие подкаталоги с приложениями, приложения, описанные соответствующим элементом <Context>, будут инициализированы согласно указанным настройкам, а все остальные приложения будут инициализированы с настройками принятыми по умолчанию.

Все WAR-файлы, обнаруженные хостом в своем корневом каталоге будут распакованы в подкаталоги с такими же именами, что и WAR-файлы, и опять же будут рассматриваться сервером как приложения-контексты. WAR-файл с именем ROOT.war (большими буквами имя, маленькими - расширение) будет распакован и инициализирован как главный контекст.

Если в корневом каталоге уже присутствует каталог с именем, схожим с WAR-файлом в этом же каталоге, WAR-файл не распаковывается. Если у вас появился новая версия WAR-файла, вам надо остановить Tomcat, удалить в корневом каталоге распакованный подкаталог с нужным приложением, положить новый WAR-файл в корневой каталог (переписав его поверх старого), и снова запустить Tomcat.

Tomcat 4 не поддерживает функцию hot deployment, как в некоторых других более мощных серверах приложений, когда для активизации приложения не нужно перезагружать Tomcat, а достаточно просто положить новый WAR-файл в корневой каталог.

Элемент <Connector>

А сейчас мы рассмотрим элемент <Connector>. Этот элемент описывает компонент, который принимает запросы, поступающие на сервер, и отправляет ответы. Элемент <Connector> соответствует интерфейсу `org.apache.catalina.Connector`. Ниже описаны допустимые атрибуты данного элемента.

className: Полное название класса, который реализует интерфейс `org.apache.catalina.Connector`. Как правило это `org.apache.catalina.connector.http.HttpConnector`.

enableLookups: Указывает, должен ли Connector обращаться к DNS, чтобы определить имя удаленного сервера. Значение по умолчанию равно "true". Если эта функция включена, вы можете узнавать в своем приложении доменное имя удаленного сервера с помощью метода `request.getRemoteHost()`. Включение данной функции может сказаться на производительности Tomcat-а. Поэтому в большинстве случаев данный атрибут должен быть равен "false".

redirectPort: указывает TCP/IP порт, на который должен быть переадресован запрос, если для его обработки требуется SSL-соединение. Ресурсы, требующие SSL-соединение описываются в web.xml файле каждого конкретного приложения-контекста.

scheme: Указывает, какой протокол связи будет использоваться для данного элемента <Connector>. По умолчанию равен "http". Для SSL-соединения присвойте этому атрибуту значение "https".

secure: Указывает, должен ли данный элемент <Connector> поддерживать SSL-соединение. По умолчанию значение равно "false".

Элемент <Connector> должен располагаться внутри элемента <Service> и ДО элемента <Engine>.

Давайте теперь более подробно рассмотрим компонент HttpConnector.

HttpConnector

Чаще всего в элементе <Connector> используется класс `org.apache.catalina.connector.http.HttpConnector`. Как и все другие родственные ему компоненты, он реализует интерфейс `org.apache.catalina.Connector` и наследует от него все те атрибуты, что мы перечислили выше. Однако, он также имеет еще и дополнительные атрибуты, которые характерны только для `HttpConnector`. Эти атрибуты перечислены ниже.

port: Указывает TCP/IP порт, который будет "прослушивать" данный элемент <Connector>.

> **На заметку**. Сразу после установки Tomcat настроен на порт 8080. это значит, что для работы с Tomcat-ом, в броузере вам следует набрать следующий URL:
>
> `http://<server>:8080/`
>
> Чтобы переключить Tomcat на порт :80, на котором обычно "сидят" web-сервера, поменяйте значение атрибута "port" на "80". После перезапуска Tomcat-а, вы сможете обращаться к нему с помощью следующего URL без указания номера порта:
>
> `http://<server>/`

> **Внимание:** Если вы переключаете Tomcat на какой-либо другой порт, убедитесь, что этот порт не занят другим процессом или другим Web-севрером (IIS или Apache). В противном случае, вы не сможете запустить Tomcat.

address: Этот атрибут используется в том случае, если у машины, на которой работает Tomcat, имеются несколько IP-адресов. Данный атрибут указывает, к какому из этих IP-адресов привязывается Tomcat. Если этот атрибут отсутствует, Tomcat будет привязан ко всем IP-адресам, имеющимся у данной машины.

bufferSize: Определяет размер буфера, который будет выделен данному компоненту для обработки входящих потоков. Увеличивая размер буфера, вы сможете повысить производительность сервера, но это в свою очередь повысит требование к объему свободной памяти. По умолчанию этот атрибут равен "2048", т.е. размер равен 2048 байтам.

className: Имя класса, реализующего интерфейс `org.apache.catalina.Connector`. В нашем случае это класс `org.apache.cataline.connector.http.HttpConnector`.

enableLookups: То же, что и для всех других элементов <Connector>.

proxyName: Указывает имя proxy-сервера, если Tomcat расположен за сетевым экраном (firewall). Этот атрибут необязателен.

proxyPort: Указывает, какой порт надо использовать, если Tomcat расположен за сетевым экраном (firewall). Этот атрибут необязателен.

minProcessors: Указывает минимальное количество обработчиков запросов, которые должен Tomcat создать при старте. По умолчанию создается 5 обработчиков.

maxProcessors: Указывает максимальное количество обработчиков запросов, которые Tomcat-у разрешается создать. По умолчанию предел равен 20 обработчикам. Если данному атрибуту

присвоить любое значение меньше нуля, Tomcat-у будет разрешено создавать неограниченное количество обработчиков.

acceptCount: Указывает, сколько запросов максимально должно находиться в очереди. По умолчанию значение равно 10.

> **На заметку**. Tomcat обрабатывает запросы достаточно быстро. Но если их поступает слишком много, а число обработчиков ограничено и очередь переполнена, пользователи в ответ на попытку соединения с Tomcat-ом получат ответ "Server is too busy" ("Сервер слишком занят"). Чтобы не разочаровывать пользователей этим сообщением, увеличьте количество обработчиков и размер очереди, но не забывайте, что для этого Tomcat потребует больше памяти.

connectionTimeout: Указывает, через какое время соединение должно быть разорвано. По умолчанию атрибут равен 60000, т.е. 60000 миллисекундам. Чтобы отключить разрыв соединения, присвойте данному атрибуту значение "-1".

Элемент <Connector> в примере, что мы привели выше, выглядит следующим образом:

```
<Connector className="org.apache.catalina.connector.http.HttpConnector"
  port="8080"
  minProcessors="5"
  maxProcessors="75"
  enableLookups="true"
  redirectPort="8443"
  acceptCount="10"
  debug="0"
  connectionTimeout="60000"/>
```

Элемент описывает, что компонент `HttpConnector` будет "слушать" порт 8080. Если на этот порт поступит запрос к ресурсу, требующему SSL-соединения, запрос будет перенаправлен на порт "8443", который "прослушивает" другой экземпляр компонента `HttpConnector` с особыми настройками для SSL-соединения. <Connector> при запуске инициализирует 5 обработчиков запросов, и ему разрешено увеличить их число максимум до 75 обработчиков.

Использование SSL

Lastly, we can declaratively control the level of security in the transport mechanism using the following tag in `web.xml`:

```
<user-data-constraint>
  <description>SSL not required</description>
  <transport-guarantee>NONE</transport-guarantee>
</user-data-constraint>
```

There are three possible values for the `<transport-guarantee>`:

Transport	Description
`NONE`	No encryption is required (http is fine)
`CONFIDENTIAL`	The data must be encrypted, so that other parties can not observe the contents (e.g. enforce SSL)
`INTEGRAL`	The data must be transported so that the data cannot be changed in transit. Most servers use SSL for this value too, although in theory you could use some hashing algorithm as

	encryption is not required

Защита ресурсов с помощью Security Realm

При защите ресурсов в веб-приложениях мы чаще всего сталкиваемся с двумя понятиями: аутентификация (authentication) и авторизация (authorization). Аутентификация – это механизм определения, что пользователь является именно тем, кто он есть. Авторизация – это механизм определения прав пользователя, чья личность подверждена путем аутентификации, на доступ и использование того или иного ресурса.

В следующей главе ("Аутентификация" - стр. 22) мы поговорим о том, какие средства предоставляет Tomcat для идентификации пользователя.

А в этой главе мы поговорим о том, как защищать ресурсы и задавать, какому пользователю какие ресурсы могут быть доступны.

Что такое Security Realm

Для защиты ресурсов веб-приложения от несанкцианированного доступа в Tomcat используется понятие "механизм безопасности" (security realm). С помощью этого механизма вы можете определить ограничения для пользователей на доступ к тем или иным файлам или ресурсам. Этот механизм встроен в Tomcat и он характерен только для Tomcat. В других серверах Java приложений – более сложных и продвинутых – используются другие механизмы.

Компонент, который обеспечивает защиту, реализует интерфейс `org.apache.catalina.Realm`. Реализация этого интерфейса определяет каким образом и какими средствами будет осуществлятся защита. В Tomcat предусмотрено, что каждый пользователь имеет имя, пароль и роли, на основании которых определяются права доступа к тому или иному файлу, каталогу или ресурсу.

Выбор механихма защиты и его настройка осуществляется с помощью элемента <Realm> в файле `server.xml`. В зависимости от того, в каком из элементов помещен элемент <Realm>, защита действует только на определенный контекст (<Context>), либо на весь хост и все контексты этого хоста (<Host>), либо на весь контейнер (<Engine>).

С Tomcat 4 в комплекте поставляются два класса, реализующих механизм защиты двумя различными способами.

Защита с помощью MemoryRealm

Первым классом, реализующим механизм защиты, является класс `org.apache.cataline.realm.MemoryRealm`. Этот класс использует простой XML-файл, в котором перечислены все пользователи, их пароли, и роли, им назначенные. При запуске Tomcat этот файл считывается в память, отсюда и вытекает название класса `MemoryRealm`. XML-файл выглядит примерно следующим образом.

```
<tomcat-users>
<user name="tomcat" password="tomcat" roles="tomcat" />
<user name="role1" password="tomcat" roles="role1" />
<user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

Внимание. В установке Tomcat по умолчанию для всех приложений используется класс `MemoryRealm`, а база пользователей для него располагается в файле `<tomcat_home>/conf/tomcat-users.xml`. Но вы можете для каждого приложения определить свой механизм защиты и свою отдельную базу пользователей. Для этого необходимо поменять некоторые атрибуты элемента <Realm> в файле `server.xml`. О том, как это сделать, мы расскажем ниже.

Как видите, в приведенном выше файле нет ничего особенно сложного. Корневым элементом файла являестя элемент `<tomcat-users>`, в котором располагается несколько элементов `<user>`. Каждый элемент `<user>` содержит всю необходимю информацию о пользователе: имя, пароль, и роли. Ниже в таблице подробно объяснено назначение каждого атрибута.

Таблица 1. Атрибуты элемента `<user>`

Атрибут	Описание
`name`	Этот атрибут содержит имя пользователя.
`password`	Этот атрибут содержит пароль.
`roles`	В этом атрибуте перечислены роли, которые присвоены данному пользователю. Пользователь получит доступ к ресурсу, елси он правильно наберет свое имя и пароль, и если одна из ролей, присвоенных ему, совпадет с ролью, упомянутой в элементе `<role-name>` в файле `web.xml` данного приложения. Если пользователю присваивается несколько ролей, они должны быть перечислены через запятую.

Давайте теперь разберемся подробнее, как работает `MemoryRealm`. Давайте создадим простое веб-приложение под названием `/onjava`, и защитим его паролем. Вот что для этого надо сделать.

1. Откройте файл `<tomcat_home>/conf/server.xml` и уберите комментарии вокруг элемента:

   ```
   <Realm className="org.apache.catalina.realm.MemoryRealm" />
   ```

   Убрав комментарии вы сделали данный механизм `MemoryRealm` действующим по умолчанию для всего контейнера. Если вы не можете найти этот элемент в своем файле server.xml, добавтье его самостоятельно сразу же под элементом <Engine>.

2. В каталоге `<tomcat_home>/webapps/` создайте подкаталог `onjava`. Это будет каталог нашего приложения. В нем создайте всю структуру подкаталогов и файлов, которая необходима для работы простейшего веб-приложения (см. главу "").

3. Откройте файл `web.xml` приложения `onjava` и добавьте в конец файла следующие элементы:

   ```
   <security-constraint>
       <web-resource-collection>
           <web-resource-name>OnJava Application</web-resource-name>
           <url-pattern>/*</url-pattern>
       </web-resource-collection>
       <auth-constraint>
           <role-name>onjavauser</role-name>
       </auth-constraint>
   </security-constraint>
   ```

   Элемент `<security-constraint>` служит для описания того, какие ресурсы вашего приложения вы хотите защитить, каким механизмом защиты вы хотите воспользоваться и

кому вы хотите разрешить доступ к защищенным ресурсам. Пока давайте рассмотрим два дочерних элемента. Первый - `<url-pattern>`. Этот элемент определяет с помощью регулярного выражения URL защищаемого ресурса. В данном примере данное регулярное приложение означает, что защищено будет все веб-приложение целиком, а не какой-то отдельный ресурс, страница или каталог. Второй дочерний элемент - `<role-name>` - определяет список ролей, которым разрешен доступ к данному защищенному ресурсу. В данном конкретном случае все веб-приложение будет доступно только тем пользователям, которым присвоена роль `onjavauser`.

4. За элементом `<security-constraint>` добавьте следующий элемент `<login-config>`:

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>OnJava Application</realm-name>
</login-config>
```

Элемент `<auth-method>` – дочерний элемент `<login-config>` – задает метод, с помощью которого у пользователя будет запрашиваться имя и пароль. Возможные варианты значений этого элемента таковы: `BASIC`, `DIGEST`, и `FORM`. О механизмах аутентификации мы поговорим ниже (см. главу "Аутентификация" – стр. 22). Элемент `<realm-name>` указывает имя ресурса, к которому данный механизм аутентификации будет применен. Это имя должно сопадать с именем, использованным в элементе `<web-resource-name>`. Это "усложнение" требуется на тот случай, если вы планируете защитить паролем несколько разных ресурсов и использовать разные механизмы аутентификации для них.

5. Откройте файл `<tomcat_root>/conf/tomcat-users.xml` и добавьте следующий элемент `<user>`:

```
<user name="bob" password="password" roles="onjavauser" />
```

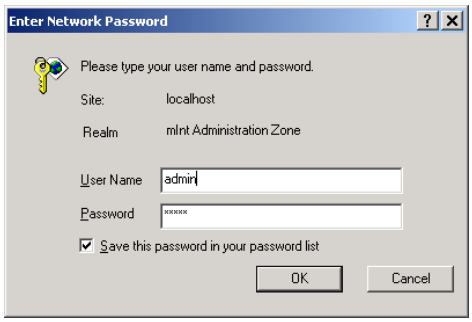
Это значит, что вы добавили нового пользователя с именем "`bob`" и паролем " `password` ". Этому пользователю присвоена роль "`onjavauser`". Эта роль совпадает с ролью, упомянутой в элементе `<security-contstraint>`. А это значит, что если пользователь правильно наберет свое имя и пароль, он получит доступ к приложению.

6. Чтобы завершить конфигурацию защиты, перезапустите Tomcat.

Давайте теперь првоерим, как работает наша защита приложения `onjava`. В адресной строке броузера наберите следующий URL:

```
http://localhost:8080/onjava/login.jsp
```

Если все было сделано правильно, вы должны увидеть на экране следующее диалоговое окно:



В поле "User Name" наберите " bob ", в поле Password – "password" и нажмите кнопку "OK". Опять же, если все было сделано правильно, вы должны получить доступ к приложению onjava.

Итак, ваше приложение защищено с помощью MemoryRealm и с использованием механизма аутентификации BASIC.

Защита с помощью JDBCRealm

The second Realm implementation provided with Tomcat is a JDBC realm. The class that implements the JDBC realm is org.apache.cataline.realm.JDBCRealm. The JDBCRealm class is much like the MemoryRealm discussed in the previous section, with the exception of where it stores its collection of users. A JDBCRealm stores all of its users in a user-defined, JDBC-compliant database. There are several steps involved when setting up a JDBC realm, but once it is configured it is really simple to manage.

Второй механизм защиты, предоставляемый Tomcat, является JDBCRealm. Полное имя класса выглядит следюущим образом: org.apache.cataline.realm.JDBCRealm. Класс JDBCRealm работает во многом так же, как и MemoryRealm, о котормо речь шла в предыдущем разделе. Исключение состоит лишь в том, что для хранения инфомрации о пользователях JDBCRealm использует не xml-файл, а базу данных. Для настройки работы JDBCRealm необходимо сделать следующее:

- Создать базу данных пользователей
- Сконфигурировать Tomcat на использование JDBCRealm
- Настроить web.xml

Для начала давайте создадим базу данных, в которой мы будем хранить сведения обо всех пользователях. В базе данных нам понадобится создат ьтри таблицы. Первая таблица будет называтсья users. В ней будут храниться имя пользователя и его пароль. Ниже перечислены названия полей в таблице users и их назначение.

Таблица 2. Поля таблицы users	
Поле	Описание
user_name	Поле user_name хранит строку с именем пользователя. Это имя пользователь будет набирать в форме. Тип поля user_name - varchar(12).
user_pass	Поле user_pass хранит строку с паролем пользователя. Тип поля user_pass - varchar(12).

Втроая таблица базы данных - roles. В таблице roles хранится список всех ролей, которые могут быть присвоены пользователям. В таблице roles только одно поле: role_name. В этом поле, чей тип - varchar(12), хранится строка с названием роли.

Третья таблица - user_roles. Таблица user_roles – связующее звено между таблицей ролей и таблицей пользователей. Ниже приводистя список полей таблицы user_roles.

Таблица 3. Поля таблицы user_roles	
Поле	Описание
user_name	Поле user_name хранит строку с именем пользователя. Типа поля user_name - varchar(12). Строка в этом поле должна соотвествовать одному из имён, упомянутых в таблице users.

role_name	Поле role_name хранит строку с названием роли, приписанной этому пользователю. Тип поля role_name - varchar(12). Строка в этом поле должна соотвествовать одной из ролей, упомянутых в таблице roles.

Содержимое кажждой из этих таблиц будет следующим.

Таблица 4. Содержимое таблицы users

user_name	user_pass
robert	password
bob	password
tomcat	password
joe	joe

Таблица 5. Содержимое таблицы roles

user_name
onjava
manager
tomcat

Таблица 6. Содержимое таблицы user_roles

user_name	user_pass
bob	onjavauser
joe	onjavauser
joe	manager
tomcat	tomcat
robert	onjavauser

Теперь, когда у нас на руках есть описание всех таблиц, мы можем присутпить к созданию базы данных tomcatusers. Для этого вам сначала надо выкачать из Интернета (по адресу http://www.mysql.com) и установить на своей машине базу данных MySQL. Кроме того, вам необходимо установить JDBC драйвер для MySQL. Его можно найти на том же самом сайте.

> **Внимание**. Для данного примера мы решили воспользоваться базой данных MySQL. Но вы можете выбрать любую другую JDBC-совместимую базу данных.

После того, как MySQL быдет установлена, выполните следующие шаги:

1. Запустите клиента базы данных mysql. Програмам расположена в каталоге <mysql_home>/bin/.
2. Создайте базу данных пользователей tomcatusers, выполнив следующую команду:

```
create database tomcatusers;
```

3. Создайте таблицу `users` с помощью следующей команды:

```
create table users
(
    user_name varchar(15) not null primary key,
    user_pass varchar(15) not null
);
```

4. Создайте таблицу `roles` с помощью следующей команды:

```
create table roles
(
    role_name varchar(15) not null primary key
);
```

5. Создайте таблицу `user_roles` с помощью следующей команды:

```
create table user_roles
(
    user_name varchar(15) not null,
    role_name varchar(15) not null,
    primary key(user_name, role_name)
);
```

6. Внесите данные в таблицу `users` с помощью следующих команд:

```
insert into users values("bob", "password");
insert into users values("joe", "$joe$");
insert into users values("robert", "password");
insert into users values("tomcat", "password");
```

7. Внесите данные в таблицу `roles` с помощью следующих команд:

```
insert into roles values("onjavauser");
insert into roles values("manager");
insert into roles values("tomcat");
```

8. Внесите данные в таблицу `user_roles` с помощью следующих команд:

```
insert into user_roles values("bob", "onjavauser");
insert into user_roles values("joe", "onjavauser");
insert into user_roles values("joe", "manager");
insert into user_roles values("robert", "onjavauser");
insert into user_roles values("tomcat", "tomcat");
```

Теперь, когда у вас сформирвоана база бользователей, давайте настроим компонент `JDBCRealm` в Tomcat-е на работу с ней. Для этого необходимо сделать следующее:

1. Откройте файл `<tomcat_home>/conf/server.xml` и закомментируйте эелемент <Realm>, котоырй мы создали в предыдущем разделе.

```
<!-- <Realm className="org.apache.catalina.realm.MemoryRealm" /> -->
```

2. Сразу же за этим элементом поместите другой элемент <Realm>:

```
<Realm className="org.apache.catalina.realm.JDBCRealm" debug="99"
    driverName="org.gjt.mm.mysql.Driver"
    connectionURL="jdbc:mysql://localhost/tomcatusers?user=test&password=test"
    userTable ="users" userNameCol ="user_name" userCredCol ="user_pass"
    userRoleTable ="user_roles" roleNameCol ="role_name"/>
```

Убедитесь, что JAR-файл, в котором содержится JDBC-драйвер базы данных, и который упоминается в атрибуте `driverName`, находится в `CLASSPATH` и виден Tomcat-у. Если вы вместо JDBC-драйвера пользуетесь JDBC-ODBC-драйвером, он тоже должен быть расположен в `CLASSPATH`. В атрибуте `connectionURL` вы должны указать имя базы данных, имя пользователя и пароль, которые будут использованы Tomcat-ом для подключения к БД. Впрочем, имя пользователя и пароль можно указать и отдельно – сотвественно в атрибутах `connectionName` и `connectionPassword`.

Ниже представлено описание других допустимых атрибутов элемента <Realm> при использовании компонента `JDBCRealm`.

Таблица 7. Атрибуты элемента `<Realm>` при использовании `JDBCRealm`

Атрибут	Описание
`classname`	Полное имя класса реализуюущего `Realm`.
`driverName`	Имя драйвера, с помощью которого Tomcat будет подключаться к базе данных пользователей.
`connectionURL`	URL-строка для связи с базой данных.
`connectionName`	Имя пользователя, которое используется для подключения к базе. Если вы пользуетесь MySQL, то это имя пользователя можно указать прямо в атрибуте `connectionURL`.
`connectionPassword`	Пароль, который используется для подключения к базе. Опять-таки, если вы пользуетесь MySQL, то это пароль можно указать прямо в атрибуте `connectionURL`
`userTable`	Название таблицы, в которой содержится списко пользователей.
`userNameCol`	Поле в таблице `userTable`, в котором хранится имя пользователя.
`userCredCol`	Поле в таблице `userTable`, в котором хранится пароль пользователя.
`userRoleTable`	Имя таблицы, в котором хранится информация о првязке пользователей к ролям.
`roleNameCol`	Поле в таблицы `userRoleTable`, в котором указана роль пользователя.

3. Чтобы завершить конфигурацию, остановите Tomcat и запустите его снова.

Внимание. Если вы решите поменять пароль пользователя, в случае `JDBCRealm` изменения вступают в силу сразу же после того, как пользователь завершил текущую сессию работы с приложением. Перезапускать Tomcat не требуется. В случае же с `MemoryRealm` при смене пароля требуется перезагрузить Tomcat.

Теперь, для проверки работы `JDBCRealm` откройте в броузере веб-приложение `onjava`. На экране должно появиться диалоговое окно уже знакомое вам по предыдущему разделу.

Аутентификация

В предыдущей главе мы познакомились с тем, как защитить ресурсы приложения от несанкционированного доступа и предоставить доступ к определенным ресурсам только тем

пользователям, которым на это дано право. Теперь давайте поговорим о том, каким образом можно удостовериться что пользователя является именно тем, за кого себя выдает.

Как правило в программных приложениях аутентификация осуществляется с помощью комбинации имени пользователя и секретного пароля, известного этому пользователю и только ему. Для проверки комбинации имени и пароля на экран выводится диалоговое окно, в которое пользователь должен ввести User Name (имя пользователя) и Password (пароль). Если комбинация совпадает с той, что известна системе, личность пользователя считается удостоверенной и он получает доступ к своим ресурсам, в противном случае пользователю предлагается повторить попытку несколько раз, после чего выводится сообщение об ошибке.

Подобный алгоритм аутентификации можно запрограммировать в веб-приложении самом с помощью пары форм и скриптов. Но эта задача настолько востребована, что во многих веб-серверах механизм аутентификации встроен изначально. Это относится и к Tomcat.

In this article, we will walk through the various security settings that we can set up in the Web Application framework, going into detail on how you can set up FORM-based authentication.

All of the code from this article should work with any Web container that supports the Servlet 2.2 API and above. This article assumes that you have knowledge of Web applications, servlets, and JSP's.

We will walk through the following items:

- Configuring security on a resource
- The various authentication options
- Using FORM-based authentication
- Enforcing SSL

Configuring security on a resource

To secure a resource we will:

- Restrict resources based on a given URL pattern
- Name the security roles that are allowed access to the resources
- Name all of the security roles in the Web application
- Name all of the users/groups in the roles

Step One: Restrict resources based on a given URL pattern

First of all, we want to protect some resource in our Web container. We restrict an area of our site based on a URL pattern. So, let's restrict access to any URL that starts with `/secure`.

All of our configuration will take place in a file named `web.xml` that lives in the magical directory `WEB-INF`. This conforms to the Web Application standard defined by Sun (and company). In your `web.xml` file you will tell the Web container that you want to restrict an area based on the URL pattern, which will look like:

```
<security-constraint>
<web-resource-collection>
  <web-resource-name>SecurePages</web-resource-name>
  <description>Security constraint /secure</description>
  <url-pattern>/secure/*</url-pattern>
  <http-method>POST</http-method>
  <http-method>GET</http-method>
</web-resource-collection>
```

The interesting tags here are:

- `<url-pattern>`: If a client accesses a URL that matches this pattern (in this case, accessing any resource under `/secure`), the container will make sure the user is authenticated before granting access to the resource
- `<http-method>`: Here we declare what HTTP methods the security constraint will apply to. If we only had "GET" defined, then a POST request will not have to go through the security conditions. NOTE: If you do not specify a `<http-method>` then the security constraint will apply to all HTTP methods.

Step Two: Name the security roles that are allowed access to the resources

Now we have defined the area that we are securing, and what HTTP methods we will allow. We still need to tell the container who has access to this given resource. To do this we set up abstract security "roles" for our entire Web application, and list the roles that have access to each `<security-constraint>`. In our example we will only let users in the "admin" role have access to `/secure` (the SecurePages resource).

After the `<web-resource-collection>` we place an `<auth-constraint>` tag that tells the container "only the admin role has access to this area." For example,

```
<auth-constraint>
<description>only let the admin users login</description>
<role-name>admin</role-name>
</auth-constraint>
```

Step Three: Name all of the security roles in the web application

Later on in `web.xml` we define all of the security roles. Our example only has one role (admin), but if you imagine a real-world example where you have `/managers` and `/peons`, each with their own roles, then you would configure the "manager" and "peon" roles, but the `<auth-constraint>` for each resource will only list one role (e.g. under the `<security-constraint>` that has the pattern `/managers/*`, the `<role-name>` would be "manager").

Here is the simple `<security-role>` tag showing our only role (admin):

```
<security-role>
  <description>The Only Secure Role</description>
  <role-name>admin</role-name>
</security-role>
```

Step Four: Name all of the users/groups in the roles

So, we have listed all of our security roles, told the server that anyone in that one role is allowed access to a resource under `/secure`, but how do we set up the users that are in the given role? A "role" is just this abstract thing, but we need to tie it to the real security system. This is where we move away from the standards, and the particular server takes over.

If we are working with BEA WebLogic Server, we tie to the real users via the file `WEB-INF\weblogic.xml`. That file would have the following xml:

```
<weblogic-web-app>
  <security-role-assignment>
    <role-name>admin</role-name>
    <principal-name>system</principal-name>
```

```
  </security-role-assignment>
</weblogic-web-app>
```

We tie to the role name `admin`, and give the usernames, or groups that are part of that role. In this case, I have only granted access to `/secure` to the user "system."

To put it all together, so far we have set up the security roles for our Web application, named all of the users and groups that are part of that role, and set up a security condition: when a browser accesses `/secure`, only the users in the admin role will get through!

Authentication Options

Although we have defined the users that are allowed access to a resource, we need to tell the container how we want to authenticate the users. There are four authentication methods to choose from:

Authentication Method	Description
`BASIC`	Use HTTP basic authentication. If we used this setting then the good ole pop up window will show trying to access `/secure`.
`FORM`	It is sometimes nice to be able to build your authentication into your Web pages. With FORM-based authentication we can do this! We will focus on this mechanism in the next section.
`CLIENT-CERT`	We can use client digital certificates to authenticate against.
`DIGEST`	Use HTTP digest authentication (advanced form of `BASIC`, but hasn't caught on to much).

This is a nice feature, being able to choose the authentication mechanism at deploy-time. Let's check out form based authentication, and walk through an example of setting it up.

FORM-based Authentication

We will go through the simple steps required in setting up the standards-based FORM-based authentication.

1. Configure the `web.xml` to use FORM-based authentication
2. Build the login form

Step One: Configure the web.xml to use FORM-based authentication

Let's tell the container to use FORM-based authentication in our `web.xml` file.

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/LoginForm.html</form-login-page>
    <form-error-page>/LoginError.html</form-error-page>
  </form-login-config>
</login-config>
```

First, we specify "FORM" as the auth-method (instead of BASIC, DIGEST, or CLIENT-CERT), and then we tell the system that the Web page `LoginForm.html` has the <FORM> which will authenticate a user. If we try to access a page under `/secure` we will first have to fill out the form in `LoginForm.html` and

authenticate. If our authentication fails (we do not log in correctly as the system user), then we will be sent to `LoginError.html`.

Step Two: Build the login form

Now we build out login form. We have to follow a couple of conventions that are defined in the Servlet API specification:

- Our `<form>`'s action field must be `j_security_check`
- We must have form fields `j_username`, and `j_password` that hold the username and password to authenticate with

So, our `LoginForm.html` will simply look like:

```
<form method="POST" action="j_security_check">

  Username: <input type="text"     name="j_username"><br />
  Password: <input type="password" name="j_password"><br />
  <br />

  <input type="submit" value="Login">
  <input type="reset"  value="Reset">

</form>
```

Let's say a browser tries to access something under `/secure` in our deployed Web application. The container will do the following:

1. Save away the resource that the user was trying to access.
2. Send back the `LoginForm.html`.
3. When the user fills out the username and password and submits it back, the container tries to authenticate the user. If the auth fails the `LoginError.html` is sent back to the browser.
4. If the authenticated user is part of the admin role (e.g. system user), the original resource will be sent back to the user, otherwise the `LoginError.html` will.

And that is it! Using FORM-based authentication is easy. You configure the `web.xml` to point to the correct login form and error page, and then make sure that the form follows the conventions of using `j_security_check`, `j_username`, and `j_password`.

Embedding Tomcat Into Java Applications

04/03/2002

In this article, we'll extend our Tomcat discussions to the application level by creating a Java application that manages an embedded version of the Tomcat JSP/servlet container. Tomcat can be broken down into a set of containers, each with their own purpose. These containers are by default configured using the `server.xml` file. When embedding, you will not be using this file; therefore, you will need to assemble instances of these containers programmatically. The following XML code snippet contains the hierarchy of the Tomcat containers:

```
<Server>
  <Service>
    <Connector />
    <Engine>
      <Host>
        <Context />
      </Host>
    </Engine>
  </Service>
</Server>
```

Note: Each of the previously listed elements contains attributes to set their appropriate behaviors, but for our purposes, only the element hierarchies and relationships are important.

This is the structure that we need to create with our embedded application. The `<Server>` and `<Service>` elements of this structure are going to be implicitly created, therefore we do not have to create these objects ourselves. The steps to create the remainder of the container structure are listed below.

These are the same steps that we must perform in order to create our own embedded version of the Tomcat container:

1. Create an instance of an `org.apache.catalina.Engine`; this object represents the `<Engine>` element above and acts as a container to the `<Host>` element.
2. Create an `org.apache.catalina.Host` object, which represents a virtual host, and add this instance to the `Engine` object.
3. Now you need to create *n*-number of `org.apache.catalina.Context` objects that will represent each Web application in this `Host`.
4. Once each of your `Context`s are created, you then need to add each of the created `Context`s to the previously created `Host`. For our example, we'll create a single `Context` that will represent our `onjava` application.
5. The final step is to create an `org.apache.catalina.Connector` object and associate it with the previously created `Engine`. The `Connector` object is the object that actually receives a request from the calling client.

To create this application, we'll leverage some existing Tomcat classes that have been developed to ease this type of integration. The main class we will use is the `org.apache.catalina.startup.Embedded` class, which can be found in the `<CATALINA_HOME>/src/catalina/src/share/org/apache/catalina/startup` directory. The following source listing contains our sample application that builds these containers using the `org.apache.catalina.startup.Embedded` class.

```
package onjava;

import java.net.URL;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Deployer;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.logger.SystemOutLogger;
import org.apache.catalina.startup.Embedded;
import org.apache.catalina.Container;
```

```
public class EmbeddedTomcat {

  private String path = null;
  private Embedded embedded = null;
  private Host host = null;
  /**
    * Default Constructor
    *
    */
  public EmbeddedTomcat() {

  }

  /**
    * Basic Accessor setting the value of the context path
    *
    * @param path - the path
    */
  public void setPath(String path) {

    this.path = path;
  }

  /**
    * Basic Accessor returning the value of the context path
    *
    * @return - the context path
    */
  public String getPath() {

    return path;
  }

  /**
    * This method Starts the Tomcat server.
    */
  public void startTomcat() throws Exception {

    Engine engine = null;
    // Set the home directory
    System.setProperty("catalina.home", getPath());

    // Create an embedded server
    embedded = new Embedded();
    // print all log statments to standard error
    embedded.setDebug(0);
    embedded.setLogger(new SystemOutLogger());

    // Create an engine
    engine = embedded.createEngine();
    engine.setDefaultHost("localhost");

    // Create a default virtual host
    host = embedded.createHost("localhost", getPath()
      + "/webapps");
    engine.addChild(host);

    // Create the ROOT context
    Context context = embedded.createContext("",
      getPath() + "/webapps/ROOT");
    host.addChild(context);

    // Install the assembled container hierarchy
    embedded.addEngine(engine);
```

```
  // Assemble and install a default HTTP connector
  Connector connector =
    embedded.createConnector(null, 8080, false);
  embedded.addConnector(connector);
  // Start the embedded server
  embedded.start();
}

/**
  * This method Stops the Tomcat server.
  */
public void stopTomcat() throws Exception {
  // Stop the embedded server
  embedded.stop();
}

/**
  * Registers a WAR with the container.
  *
  * @param contextPath - the context path under which the
  *               application will be registered
  * @param warFile - the URL of the WAR to be
  * registered.
  */
public void registerWAR(String contextPath, URL warFile)
  throws Exception {

  if ( contextPath == null ) {

    throw new Exception("Invalid Path : " + contextPath);
  }
  if( contextPath.equals("/") ) {

    contextPath = "";
  }
  if ( warFile == null ) {

    throw new Exception("Invalid WAR : " + warFile);
  }

  Deployer deployer = (Deployer)host;
  Context context = deployer.findDeployedApp(contextPath);

  if (context != null) {

    throw new
      Exception("Context " + contextPath
      + " Already Exists!");
  }
  deployer.install(contextPath, warFile);
}

/**
  * Unregisters a WAR from the web server.
  *
  * @param contextPath - the context path to be removed
  */
public void unregisterWAR(String contextPath)
  throws   Exception {

  Context context = host.map(contextPath);
  if ( context != null ) {

    embedded.removeContext(context);
  }
  else {
```

```
      throw new
        Exception("Context does not exist for named path :
        + contextPath);
    }
  }

  public static void main(String args[]) {

    try {

      EmbeddedTomcat tomcat = new EmbeddedTomcat();
      tomcat.setPath("d:/jakarta-tomcat-4.0.1");
      tomcat.startTomcat();

      URL url =
        new URL("file:D:/jakarta-tomcat-4.0.1"
        + "/webapps/onjava");
      tomcat.registerWAR("/onjava", url);

      Thread.sleep(1000000);

      tomcat.stopTomcat();

      System.exit(0);
    }
    catch( Exception e ) {

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

You should begin your examination of the `EmbeddedTomcat` application source with the `main()` method. This method first creates an instance of the `EmbeddedTomcat` class. It then sets the path of the Tomcat installation that will be hosting our Tomcat instance. This path is equivalent to the `<CATALINA_HOME>` environment variable. The next action performed by the `main()` method is to invoke the `startTomcat()` method. This is the method that implements the container-construction steps described earlier. The steps performed by this method are listed below.

1. The `main()` method begins by setting the system property to the value of the path attribute:

```
2.    // Set the home directory
      System.setProperty("catalina.home", getPath());
```

Note:

Make sure you use the value of `<CATALINA_HOME>` as the directory value passed to the `setPath()` method.

---

3. The next step performed by this method is to create an instance of the `Embedded` object and set the debug level and current logger.

```
4.    // Create an embedded server
5.    embedded = new Embedded();
6.    embedded.setDebug(5);
7.    // print all log statments to standard error
```

```
embedded.setLogger(new SystemOutLogger());
```

Note:

The debug level should be `0`, when deploying a production Web application. Setting the debug level to `0` reduces the amount of logging performed by Tomcat, which will improve performance significantly.

---

8. After the application has an instance of the `Embedded` object, it creates an instance of an `org.apache.catalina.Engine` and sets the name of the default host. The `Engine` object represents the entire Catalina servlet container.

```
9.   // Create an engine
10.    engine = embedded.createEngine();
     engine.setDefaultHost("localhost");
```

11. After an `Engine` has been instantiated, we create an `org.apache.catalina.Host` object, named `localhost`, with a path pointing to the `<CATALINA_HOME>/webapps/` directory, and add it the `Engine` object. The `Host` object defines the virtual hosts that are contained in each instance of a Catalina Engine.

```
12.    // Create a default virtual host
13.    host = embedded.createHost("localhost", getPath() +
14.      "/webapps");
15.
     engine.addChild(host);
```

16. The next step performed by the `startTomcat()` method is to create an `org.apache.catalina.Context` object, which represents the `ROOT` Web application packaged with Tomcat, and add it the to the previously created `Host`. The `ROOT` Web application is the only application that will be installed by default.

```
17.    // Create the ROOT context
18.    Context context = embedded.createContext("",
19.      getPath() + "/webapps/ROOT");
       host.addChild(context);
```

20. The next step adds the `Engine` containing the created `Host` and `Context` to the `Embedded` object.

```
21.    // Install the assembled container hierarchy
     embedded.addEngine(engine);
```

22. After the engine is added to the `Embedded` object, the `startTomcat()` method creates an `org.apache.catalina.Connector` object and associates it with the previously created `Engine`. The `<Connector>` element defines the class that does the actual handling of requests and responses to and from a calling client application. In the following snippet, an HTTP connector that listens to port 8080 is created and added to the `Embedded` object.

```
23.    // Assemble and install a default HTTP connector
24.    Connector connector = embedded.createConnector(null,
25.      8080, false);
     embedded.addConnector(connector);
```

26. The final step performed by the `startTomcat()` method starts the Tomcat container.

```
     embedded.start();
```

When `startTomcat()` returns, the main method calls the `registerWAR()` method, which installs the previously deployed `onjava` application to the `Embedded` object. The URL used in this example can point to any Webapp directory that follows the specification for Java Servlet 2.2 and later.

```
URL url =
  new URL("file:D:/jakarta-tomcat-4.0.1"
  + "/webapps/onjava");
tomcat.registerWAR("/onjava", url);
```

The main application is then put to sleep to allow the embedded server time to service requests. When the application awakes, the embedded server is stopped and the application exits.

To test this application, you must complete the following steps:

1. Compile the `EmbeddedTomcat.java` class.
2. Make sure all other instances of Tomcat are shut down.
3. Add the following *jar* files, all of which can be found in the Tomcat installation, to your application classpath.
    - <CATALINA_HOME>/bin/bootstrap.jar
    - <CATALINA_HOME>/server/lib/catalina.jar
    - <CATALINA_HOME>/server/lib/servlet-cgi.jar
    - <CATALINA_HOME>/server/lib/servlets-common.jar
    - <CATALINA_HOME>/server/lib/servlets-default.jar
    - <CATALINA_HOME>/server/lib/servlets-invoker.jar
    - <CATALINA_HOME>/server/lib/servlets-manager.jar
    - <CATALINA_HOME>/server/lib/servlets-snoop.jar
    - <CATALINA_HOME>/server/lib/servlets-ssi.jar
    - <CATALINA_HOME>/server/lib/servlets-webdav.jar
    - <CATALINA_HOME>/server/lib/jakarta-regexp-1.2.jar
    - <CATALINA_HOME>/lib/naming-factory.jar
    - <CATALINA_HOME>/common/lib/crimson.jar
    - <CATALINA_HOME>/common/lib/jasper-compiler.jar
    - <CATALINA_HOME>/common/lib/jasper-runtime.jar
    - <CATALINA_HOME>/common/lib/jaxp.jar
    - <CATALINA_HOME>/common/lib/jndi.jar
    - <CATALINA_HOME>/common/lib/naming-common.jar
    - <CATALINA_HOME>/common/lib/naming-resources.jar
    - <CATALINA_HOME>/common/lib/servlet.jar
    - <CATALINA_HOME>/common/lib/tools.jar
4. Make sure that your classpath includes the directory containing the compiled `EmbeddedTomcat` class.
5. Execute the following command:

```
java onjava.EmbeddedTomcat
```

If everything went according to plan, you should see some log statements in the console window:

```
HttpProcessor[8080][0] Starting background thread
HttpProcessor[8080][0]  Background thread has been started
HttpProcessor[8080][1] Starting background thread
HttpProcessor[8080][1]  Background thread has been started
HttpProcessor[8080][2] Starting background thread
HttpProcessor[8080][2]  Background thread has been started
HttpProcessor[8080][3] Starting background thread
HttpProcessor[8080][3]  Background thread has been started
HttpProcessor[8080][4] Starting background thread
```

```
HttpProcessor[8080][4]  Background thread has been started
```

Once you see the previous text, you will be able to access the `ROOT` and `/onjava` Web applications using the following URLs:

- http://localhost:8080/
- http://localhost:8080/onjava/

Note: The `onjava` application that we are using throughout this article is the Web application from my previous Tomcat articles.

Up next: in the next Tomcat article, we will continue our embedded discussions by debugging a Web application that is running in our embedded container.

Примерный план

Chapter 6: Web Application Administration
Chapter 7: Manager Configuration
Chapter 8: Advanced Standard Features
Chapter 9: Class Loaders
Chapter 10: HTTP Connectors
Chapter 11: Connectors
Chapter 12: The WARP Connector
Chapter 13: The AJP Connector
Chapter 14: IIS
Chapter 15: JDBC
Chapter 16: Security
Chapter 17: Additional Uses for Ant
Chapter 18: Log4J
Chapter 19: Shared Tomcat Hosting
Chapter 20: Server Load Testing