Tomcat Administration for Linux

with examples and
hands-on exercises

WEBUCATOR

Version: 3.1.2

Class Files

Download the class files used in this manual at
https://static.webucator.com/media/public/materials/classfiles/TOM201-3.1.2.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Table of Contents

LESSON 1
Tomcat Introduction

Topics Covered

☑ The history of Tomcat.

☑ The version features of Tomcat.

☑ The major components of Tomcat.

☑ The fundamentals of JEE architecture.

Introduction

In this lesson you will learn the evolution of Tomcat from version 3 to version 10. In addition, important components such as Catalina, the servlet container, will be presented. At the end of the lesson, you will study JEE architecture and the MVC design pattern and learn how these technologies dictate the structure of a web application.

1.1. History of Tomcat

Tomcat was originally developed by Sun Microsystems as a reference implementation for a Java servlet container. The code for Tomcat is written in Java and therefore requires the services of the **Java Virtual Machine** (JVM). Because JVMs are available for many Unix, Linux, Windows and enterprise platforms, Tomcat can be installed on a wide range of servers.

Sun donated Tomcat to the Apache Software Foundation (ASF) after intensive research and development on the container. When Tomcat became available in 1999, the product was the first major servlet container and enabled administrators and developers to write Java server side code. At the time, the majority of server side web development was accomplished utilizing PERL scripts on Unix machines and ActiveServer Pages on Microsoft computers.

For each version of Tomcat, one or more releases are published to address bugs, provide enhancements, and introduce new features. Of particular concern was the need for a durable servlet engine. A servlet engine manages the lifecycle of a servlet. Accordingly, Tomcat version 4 introduced Catalina, a big step from version 3.

Java was also enhanced through successive releases, although the product remains in version 1. Nonetheless, big steps have been taken across product releases. For example, Java version 1.2 introduced Enterprise Edition, a profound development in the evolution of Java and the introduction of the web server side coding in the Java programming language.

The following table shows the features introduced in Tomcat since version 3. For clarity, incremental "releases" (modifications to releases, e.g. 3.1.1) are omitted. Minimum Java version and release indicates the earliest JVM acceptable to run the Tomcat version and release.

1.2. Version Number and Features

Tomcat Version and Features

Tomcat Version and Release Number	Comments	Minimum Java Version and Release Number
3.0	Initial Release	1.1
3.1	Servlet reloading, WAR file support	1.1
3.2	Refactoring of internals	1.1
3.3	Performance improvements over previous releases	1.1
4.0	Introduced the Catalina servlet container	1.1
4.1	JMX support, Coyote HTTP/1.1 connector introduced	1.3
5.0	Performance enhancements including reduced garbage collection, expanded JMX capabilities	1.3
5.5	Stability and performance improvements	1.4
6.0	Memory optimizations, Nonblocking I/O, refactored clustering	1.5
7.0	Servlet 3.0 and JSP 2.2	1.6
8.0	Servlet 3.1 and JSP 2.3	1.7
8.5	HTTP/2, OpenSSL for JSSE, TLS Virtual Hosts	1.7
9.0	Servlet 4	1.8
10.0	Jakarta Servlet 5.0, Jakarta Server Pages 3.0, Jakarta WebSocket 2.0	1.8

1.3. Tomcat Components

Tomcat is comprised of many components that are configured in the `server.xml` file. The major components are discussed below.

❖ 1.3.1. Catalina

Catalina is the servlet engine. The engine is responsible for executing servlets and managing the servlet lifecycle. Catalina is a Java class that implements Oracle's servlet specification.

❖ 1.3.2. Jasper

Jasper is responsible for converting JSPs (Java server pages) into servlets. When a JSP page is requested, Jasper checks the modification timstamp of the file and will convert the JSP into a servlet if the timestamp is more recent than the timestamp of the generated servlet.

❖ 1.3.3. Coyote

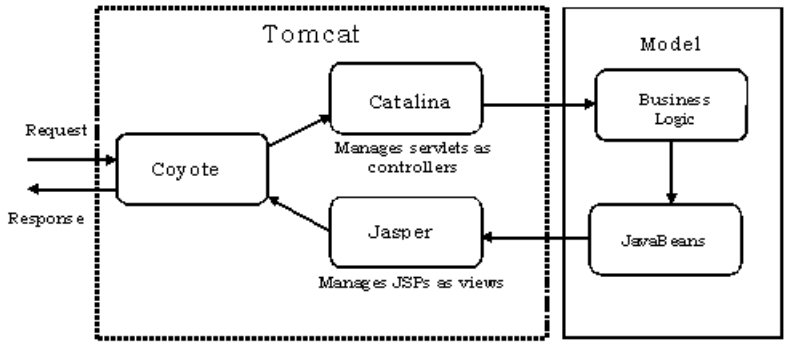
Coyote is the HTTP protocol connector that listens for requests originating from clients and sends responses back to clients.

1.4. JEE Overview

JEE (Java Enterprise Edition) is a platform of technologies that conform to various JSRs (Java Specification Requests). The following list is representative of the major platforms:

- Servlet
- JSP
- Web Services
- Enterprise JavaBeans 3
- Java Persistence 2
- Java Server Faces2
- Java Naming and Directory Interface

❖ 1.4.1. MVC Design Pattern



JEE web applications are structured to be compatible with the **MVC** (Model View Controller) design pattern.

- The **controller** interprets requests and decides on the course of action. In JEE, the controller is manifested as a servlet.
- The **model** is represented by JavaBeans that expose properties through getter/setter methods. JavaBeans represent entities in the business model. JavaBeans are instantiated by Java classes implementing business logic.
- The **view** is responsible for setting up the presentation of the data and sending the HTML output to the client. The view is implemented as a JSP.

Tomcat is not a JEE server. Tomcat is a servlet container. As such, however, Tomcat allows developers to build web applications pursuant to JEE specifications.

❖ 1.4.2. Servlets/JSP

Servlets represent controllers in a JEE web application. The `service` method in a servlet calls another method based on the HTTP method type. For example, the `service` method calls the `doGet` method for an HTTP GET request. In the `doGet` method, the developer can determine what values were sent in the query string and process the data accordingly.

The `service` method calls the `doPost` method for an incoming HTTP POST request. The signature of the `doGet` and `doPost` methods are identical: each method has two parameters of type `HttpServletRequest` and `HttpServletResponse`. The `HttpServletRequest` object represents

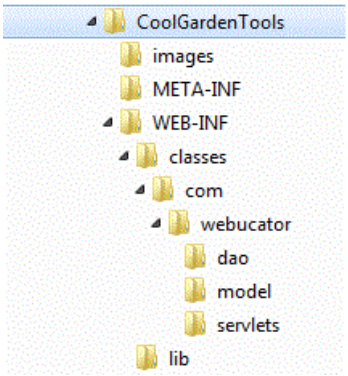
the incoming HTTP request whereas the `HttpServletResponse` object represents the outgoing HTTP response. Form-based data can be obtained by calling the `getParameter` method of `HttpServletRequest`. A response can be sent back to the client by obtaining a `java.io.PrintWriter` object from the `getWriter` method of `HttpServletResponse`. The servlet, however, is not responsible for sending a response according to the implementation of MVC in Java-based web applications. The JSP is responsible for sending responses, as we will see in the next paragraph.

JSPs represent views in the web application. A JSP page consists of JSP tags intermixed with HTML tags. JSP 2.0 introduced the JSP **EL** (Expression Language) that facilitates incorporating JavaBeans within the HTML. Although **scriptlets** containing Java code can be placed in a JSP, this is not a best practice. Instead, procedural logic can be accomplished using elements in the **JSTL** (JSP Standard Tag Library).

The servlet (controller) constructs a JavaBean object obtained from the business code layer and places the object in a `request` or `session` **attribute**. The JSP can reference this object using the EL and wrap HTML around the object's properties (e.g. customer name). Generally the resulting data is presented in an HTML **table**. The HTML is then sent to the client (**Note:** the JSP is converted into a servlet, therefore a servlet is actually performing the tasks of building a view. However, this conversion is handled by the Jasper servlet and is transparent to the developer).

❖ 1.4.3. Directory Structure

JEE web applications have a standard directory structure. The following example demonstrates this directory structure.



The context root folder is `CoolGardenTools`. By default, the context root folder name is the same as the **`context path`** that appears in the incoming URL (this can be changed in a context descriptor as you will learn in a later lesson).

The `images` directory stores image files (e.g. `.gif` and `.jpeg`). This directory is not mandated by the JEE specification but is normally included to resolve references to image files. This is also the case for other customary directories such as those that store cascading style sheets and JavaScript files.

The `META-INF` folder contains the optional `context.xml`. A context is the equivalent of a web application. The context descriptor represented by `context.xml` provides "meta-data" for the web application including data source information and document base location. This file will be copied by Tomcat to `CATALINA_BASE/conf/Catalina/localhost` and renamed to the context path (**Note:** the `localhost` directory is named after the host name appearing in the Tomcat configuration. In this course, we will perform our work on the local machine, i.e. "localhost").

The `WEB-INF` folder contains web application information. The `web.xml` descriptor file is located in this directory as well as the `classes` folder. This directory contains the Java classes, including the servlets. Java classes are stored in **packages**. A package is represented by a directory hierarchy under `classes`. In the example, one package is `com.webucator.servlets`. In the `servlets` directory we would expect to find the servlet classes.

The `WEB-INF` folder also contains the `lib` directory; jar files required by the web application are placed in this folder. A jar file represents one or more compressed Java packages.

❖ 1.4.4. JNDI

JNDI (Java Naming and Directory Interface) is an API that JEE applications can utilize to locate resources such as JDBC data sources. We will cover JNDI data sources and JDBC in a future lesson.

Conclusion

In this lesson, you have learned:

- The history of Tomcat.
- The major components of Tomcat.
- The important features of JEE architecture.

LESSON 2
Installing Tomcat

Topics Covered

☑ Installing Tomcat on Linux.

☑ Important environment variables.

☑ Starting and stopping the Tomcat Server.

☑ Verifying server operation.

Introduction

Tomcat is a part of the Apache Software Foundation and therefore is freely available. The website `tomcat.apache.org (https://tomcat.apache.org)` is devoted to Tomcat. In this lesson you will learn how to install and test the operation of the Tomcat 10.0 server.

2.1. Installation

The tar file can be downloaded at `https://tomcat.apache.org/download-10.cgi`. Extract the tar file to a directory on your file system, e.g., `/Tomcat10`.

2.2. Environment Variables

The two primary environment variables are CATALINA_HOME and CATALINA_BASE.

- The CATALINA_HOME environment variable points to the installation directory of Tomcat.
- The CATALINA_BASE points to your configuration.
- If CATALINA_BASE is assigned a directory value different from CATALINA_HOME then the directory referenced by CATALINA_BASE is presupposed to contain "dynamic" files

(e.g. `CATALINA_BASE/conf/server.xml`). In this way, you can start multiple instances of Tomcat using a common CATALINA_HOME.

- For CentOS, CATALINA_HOME and CATALINA_HOME by default are assigned to the Tomcat installation directory, e.g., `/opt/tomcat9/`.

2.3. Starting the Server

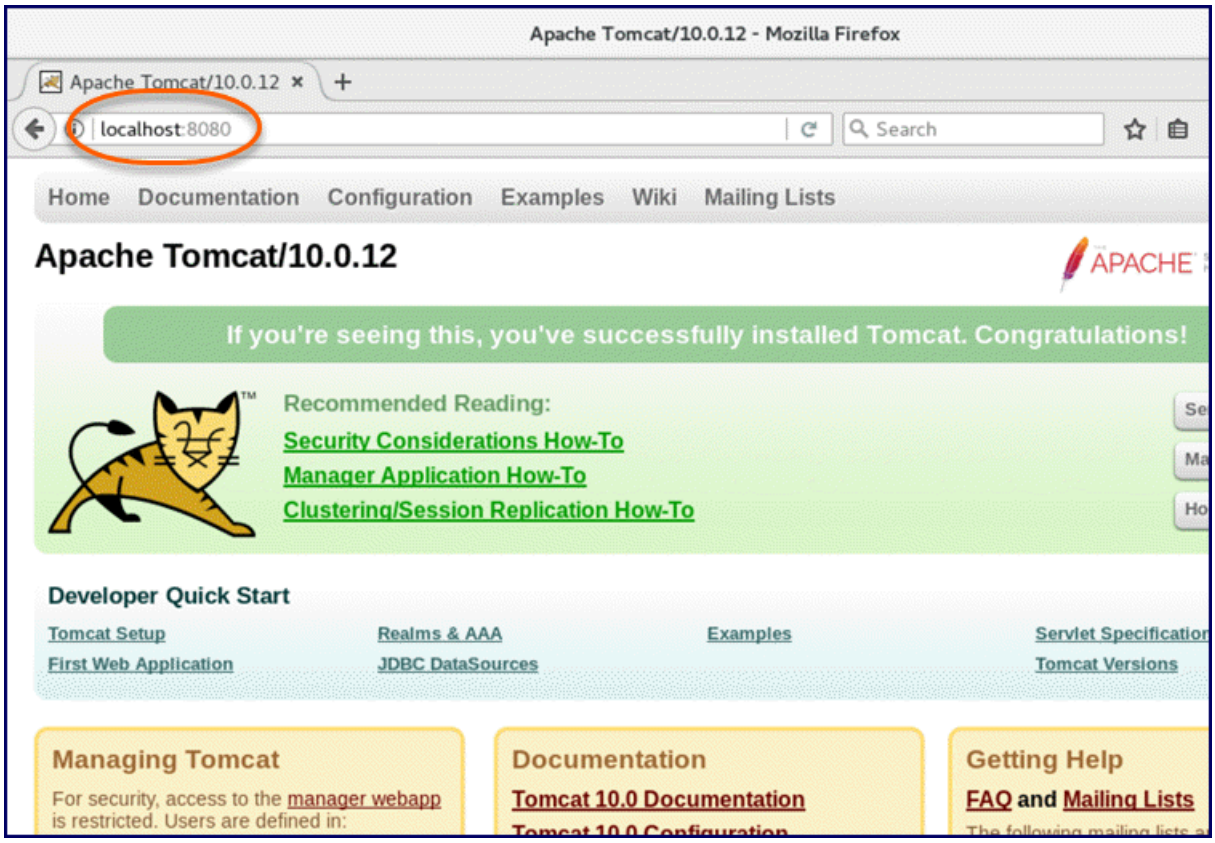
The Tomcat server is started by running a script in the `CATALINA_HOME/bin` directory. In a terminal window, navigated to this directory and then enter:


```
sudo ./startup.sh
```

The Tomcat server is now operational.

2.4. Verifying Server Operation

The server is now ready to receive requests from the browser. Go to your browser (for example, Mozilla FireFox) and type in the following: `http://localhost:8080`



2.5. Stopping the Server

To stop the Tomcat server enter the following command in the terminal window:

```
sudo ./shutdown.sh
```

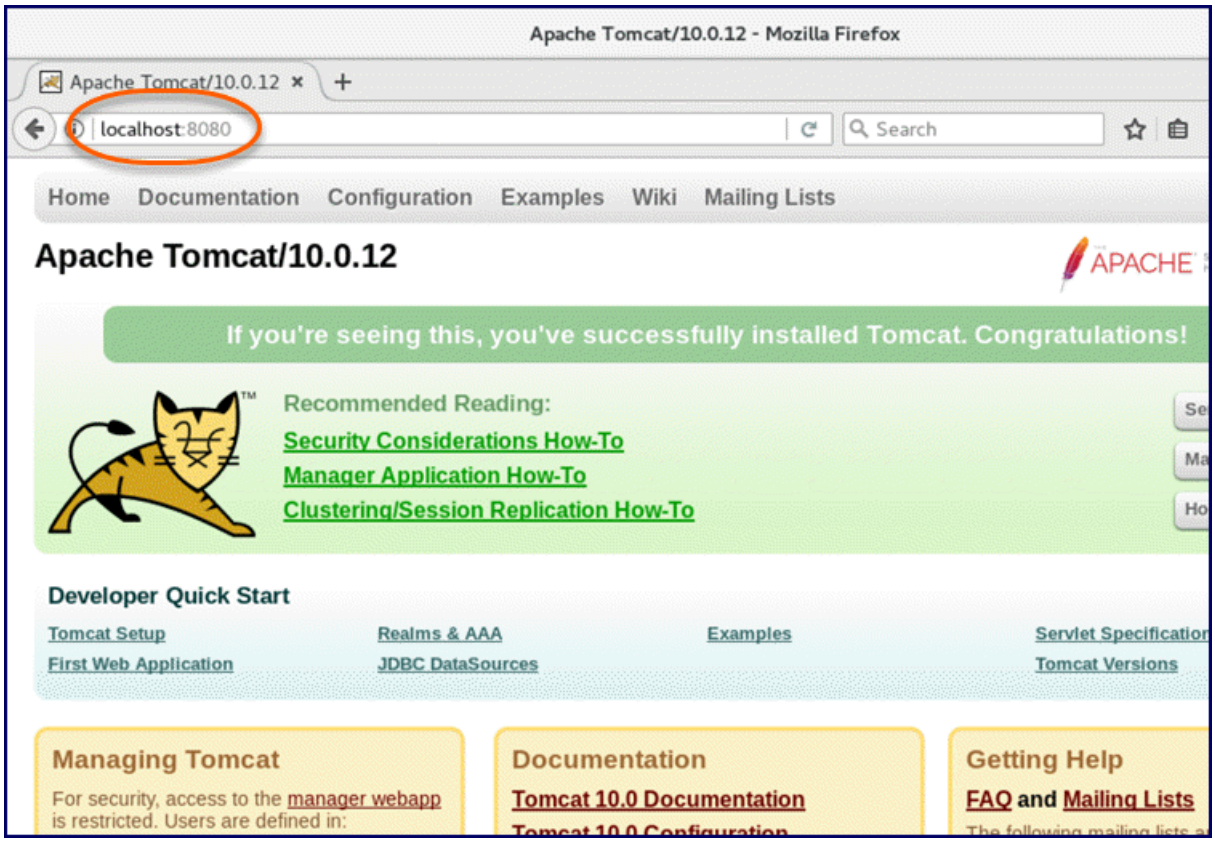
Exercise 1: Testing Tomcat

15 to 25 minutes

In this exercise you will test Tomcat by starting, stopping and restarting the Tomcat service. You will also learn how to use File Manager in super user mode.

1. Open a terminal window by clicking "Applications" in the upper lefthand corner of the screen and then selecting "Terminal".
2. Start the Tomcat server.
3. Point your browser to `http://localhost:8080`. The Tomcat welcome page should appear.
4. Stop the Tomcat server. In the browser, refresh the page. You should be notified by the browser that the server cannot be found.
5. In the terminal window, start the Tomcat server. Return to the browser to verify the welcome page can now be viewed.
6. In this course you will be required to create folders and edit files. These tasks can be accomplished in the File Manager (denoted by "Files" when you select "Applications" at the top of the screen and then "Accessories" in CentOS) if you have root privileges.
7. To start the File Manager as root select "Terminal" from the "System Tools" item in the "Applications" menu. In the new terminal window, enter `sudo nautilus`. This will open the File Manager in super user mode, allowing you to create, rename and edit folders and files.
8. In the original terminal window, you will be able to manage the Tomcat server as shown earlier and enter Linux commands as indicated throughout the training.

Solution



Conclusion

In this lesson, you have learned:

- How to install Tomcat on Linux.
- The meaning of important environment variables.
- How to start and stop the Tomcat Server.
- How to verify server operation.

LESSON 3

Tomcat Directory Structure

Topics Covered

☑ The directory structure of Tomcat.

☑ Locating the startup and shutdown batch files.

☑ About important configuration files.

☑ Where logs are stored.

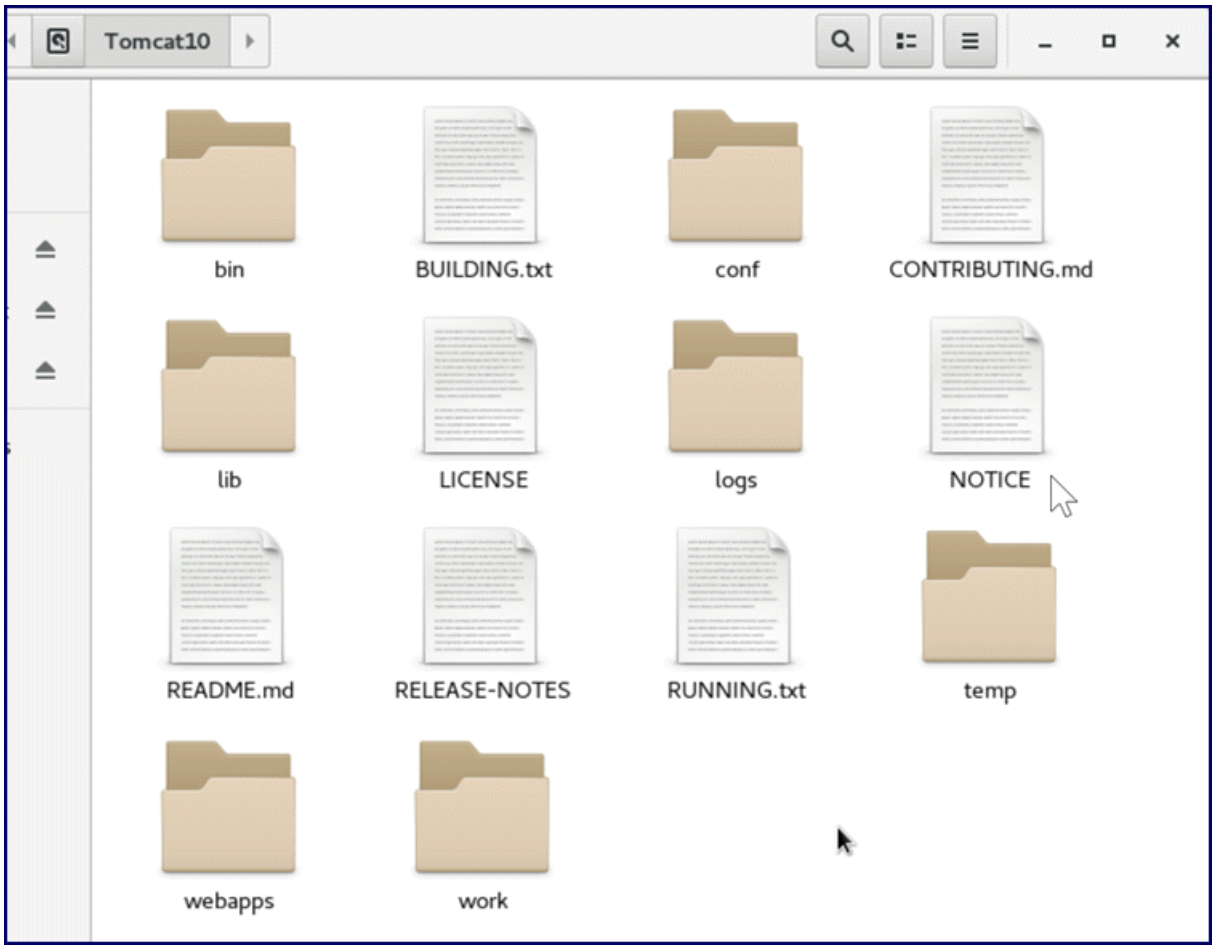
☑ Where web applications are stored.

☑ The location of jar files shared by web applications.

☑ Temporary servlet and temporary file locations.

Introduction

The Tomcat directory structure is important for the administrator to understand. Configuration files, startup and shutdown scripts, web applications, log files and other important items are located in the folder architecture.



The picture shows the Tomcat 10 directory structure in CentOS.

3.1. Batch files in /bin

The `/bin` directory contains the startup and shutdown scripts, stored as batch files. In addition, you may create the `setenv.sh` file in this directory.

Demo 3.1: LINUX-tomcat-tomcat-directory-structure/Demos/setenv.sh

```
1.    export CATALINA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxre  ↵↵
              mote.port=9009 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.man  ↵↵
              agement.jmxremote.authenticate=false"
```

Code Explanation

The above file sets the options passed to the JVM and the Catalina engine. The `setenv.sh` file, if present, is called by the Catalina startup script. In this file we can specify the value of environment variables such as `JAVA_OPTS` and `CATALINA_OPTS`. The JAVA_OPTS permit you to specify Java virtual machine options while the `CATALINA_OPTS` variable permits you to specify options to the Catalina engine. The `CATALINA_HOME` and `CATALINA_BASE` environment variables can also be set in this file.

3.2. Primary Configuration Files

❖ 3.2.1. server.xml

This file is the primary configuration file for the Tomcat server. The file contains elements that permit the administrator to adjust properties of the server, the services and associated engine as well as connectors and virtual hosts. These important attributes will be discussed in detail in a future chapter.

Demo 3.2:
LINUX-tomcat-tomcat-directory-structure/Demos/serverConnectorPortChange.xml


```
1.    <Connector port="8585" protocol="HTTP/1.1"
2.      connectionTimeout="20000"
3.      redirectPort="8443" />
```

Code Explanation

The above file contains the `Connector` element in `CATALINA_BASE/conf/server.xml` that specifies an alternate port number for the HTTP listener. This demo will illustrate setting the HTTP listener port to a value other than 8080 in `server.xml`. For example, you may have another server listening on this port, preventing Tomcat from starting.

- In the `server.xml` file, locate the occurrence of the text `8080`. The first occurrence is actually within an XML comment so repeat the find operation in order to locate the next occurrence.
- The Connector element you have located specifies the HTTP/1.1 listener port. Change '8080' to '8585'.
- After you have changed the port number, save the file Keep the editor open because you will change the port back to '8080' after testing your change.

- Shutdown the Tomcat server and then start the server.
- Test the new port number by typing the following into your browser: `http://localhost:8585`. The Tomcat welcome screen should appear.
- Return to the editor and change the port from '8585' to '8080'. Save the file.
- Test the original port number by typing the following into your browser: `http://localhost:8080`. The Tomcat welcome screen should appear.

As noted earlier, if the port you have selected is in use, Tomcat will not start properly (**Note:** although we are focusing here on the HTTP port, other unobstructed ports are required for the successful operation of Tomcat. For example, the Catalina engine listens on port 8005 for the shutdown command).

To list the ports and addresses currently in use on your computer, enter the following command in a command prompt:

```
netstat -a -n -o -p TCP.
```

A list of the address/port combinations currently active on your machine will be displayed.

❖ 3.2.2. context.xml

This file can be utilized to specify resource information for all web applications. Tomcat represents a web application as a **context**.

In a later chapter you will learn how JNDI data source information can be stored here and therefore be available to all web applications. A context descriptor can be provided for an individual web application (context). This descriptor can appear in the scope of a `Host` element within `server.xml`. A descriptor can also be presented in a stand-alone XML file. In a later lesson you will learn how to deploy a web application using a context descriptor stored in an XML file.

❖ 3.2.3. web.xml

This file is used by Tomcat to define servlet URL mappings, MIME (Multipart Internet Mail Extensions) types, welcome files. This file represents a **web application descriptor** and applies globally to all web applications.

- Servlet mappings relate a resource request to a servlet class. For example, all URLs that end with `*.jsp` are mapped to the Jasper servlet.

- Each MIME type is associated with a file extension and determines what value Tomcat will assign to `Content-Type` in the HTTP response. For example, when Tomcat serves a file with an extension of `.html`, the content type will be set to `text/html`. The `Content-Type` informs the browser what type of file has been sent by the web server.
- A welcome file is a web document sent to the client when a request does not specify a web page or a servlet URL mapping. In this case, Tomcat attempts to locate a file under the application base that matches the name of a welcome file (e.g. `index.html`). If a match is found, this page is sent to the client.

Each web application has its own descriptor file that contains definitions for welcome files, servlet URL mappings, resource references and security constraints that apply only to that context.

3.3. /logs

This is the destination of log files. Log file path and names can be specified in `CATALINA_BASE/conf/logging.properties`. Logging will be discussed in detail in a later lesson.

3.4. /webapps

This folder is the **application base** referenced in `server.xml` for the virtual host **localhost**.

- The **document base** of a web application for this host is a subfolder of `webapps` unless the web application is deployed using a context file.
- A context file is comprised of XML and contains a single `Context` element; the `docBase` attribute may specify a fully-qualified path to a web application's document base.
- Context files can be placed in `CATALINA_BASE/conf/Catalina/`*`hostName`* .
- The `Context` element may appear within the virtual host definition in `server.xml`. Multiple `Context` elements may appear in this file.

3.5. /common

This folder serves as a Java classes repository that is accessible to all web applications.

3.6. /work

The Jasper servlet converts JSPs into servlets and stores the generated source in this folder. The source is available for developers to view in case compile or logic errors resulted from the generated program.

A subdirectory is created under `/work` with an engine name (e.g. `Catalina`). A subdirectory is created beneath the engine folder for each virtual host with a name identical to the host name (e.g. `localhost`). The web application context root directories are located under the virtual host folder. For the FormDemo web application, the path under `/work` would be `CATALINA_BASE/work/Catalina/localhost/FormDemo`. The servlets generated by Jasper for JSPs in the FormDemo web application would be placed here.

Exercise 2: Getting Acquainted with the /work Directory and Generated Servlets

25 to 40 minutes

In this exercise, you inspect a servlet generated by Jasper. Jasper converts a JSP into a servlet and stores the source code and the class file in `CATALINA_HOME/work`.

This exercise will familiarize you with the `/work` directory. Later lessons will explore other Tomcat folders including `/conf`, `/log` and `/webapps`.

1. Shutdown the Tomcat server.
2. In File Manager, navigate to `CATALINA_BASE/webapps/ROOT`.
3. Rename `index.jsp` to `indexRENAMED.jsp`.
4. Copy `index.jsp` from the "Exercises" folder for this lesson to the "ROOT" directory (where `indexRENAMED.jsp` is located).
5. Navigate to `CATALINA_BASE/work`. You will see a `/Catalina` folder. Double-click this folder to view `/localhost`. This folder stores directories representing each web application. Delete all of the directories located under `localhost`.
6. Start the Tomcat server.
7. Return to the File Manager window. You should now see a folder for each deployed web application. Notice the directory named `ROOT`. This represents the `ROOT` web application. This folder is currently empty.
8. In your browser, go to `http://localhost:8080`. The date and time will be displayed. The web page for this request is `index.jsp` that you copied to the `CATALINA_BASE/webapps/ROOT` directory. The HTML output that is rendered by your browser was produced by the servlet that Jasper created from `index.jsp`.
9. In File Manager, open the `ROOT` folder. You will see the `org` directory that represents the high level portion of the package name of the generated servlet. Double-click this folder to view the `apache` directory and then drill down one more level to view the `jsp` folder. The complete package name is `org.apache.jsp`. Open the `jsp` folder and you will see the generated servlet source file and class file. The name of the generated servlet is `index_jsp`. The name is derived by appending `_jsp` to the JSP file name (minus the `.jsp` extension).

10. Open `index_jsp.java` in your text editor. This code generates the HTML that is ultimately sent to the browser. Compare the code with the web page displayed by your browser.
11. Open `CATALINA_BASE/conf/web.xml` in your text editor. Locate the URL mapping pattern `*.jsp`. Place an XML open comment tag (<!--) before `<servlet-mapping>`. Place a XML close comment tag (-->) after `</servlet-mapping>`. This will comment out the `*.jsp` mapping. Save the file.
12. Restart the Tomcat server.
13. In your browser refresh the `index.jsp` page. The display is much different now! The JSP file has been sent to the browser as plain text.
14. The Jasper servlet is inactive and therefore cannot convert the JSP to a servlet. Tomcat is sending the JSP to the browser as a plain HTML document (**Note:** the `.jsp` extension is not mapped to a MIME type in `CATALINA_BASE/conf/web.xml`. The default MIME type as dictated by the Servlet specification is `text/html` and this is enforced by Tomcat. Interestingly enough, the default MIME type cannot be overriden in `web.xml`).
15. Remove the comments that you placed around the servlet mapping in `CATALINA_BASE/conf/web.xml`. Save the changes.
16. Restart the Tomcat server.
17. Refresh the web page in your browser. The date and time should be displayed as you observed earlier. When you commented out the servlet mapping of the Jasper servlet this action prevented the servlet from converting JavaServer Pages into servlets. This modification to `web.xml` should not be performed in practice. Now you can understand why!
18. In File Manager, navigate to `CATALINA_BASE/webapps/ROOT`. Delete `index.jsp`. Rename `indexRENAMED.jsp` to `index.jsp`, reversing the renaming process you performed at the beginning of the exercise.
19. Restart the Tomcat server. Refresh the web page in your browser. You should see the original Tomcat welcome web page.

Conclusion

In this lesson, you have learned:

- The directory structure of Tomcat.
- The location of startup and shutdown batch files.
- The important configuration files.

- Where logs are stored.
- Where web applications are stored.
- The location of jar files shared by web applications.
- Temporary servlet and temporary file locations.

LESSON 4
Configuring Tomcat

Topics Covered

☑ The role of `server.xml`.

☑ The layout of the Tomcat memory address space including Server, Service, Virtual Hosts and other components.

☑ The elements in `server.xml` that configure the components.

☑ Configuring Tomcat by updating `server.xml`.

Introduction

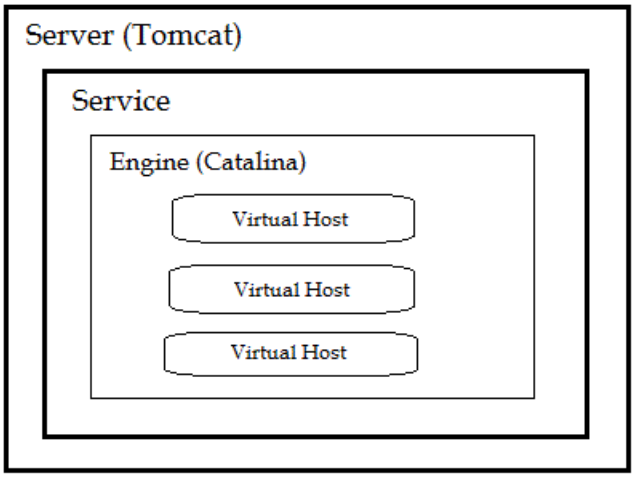
This lesson will cover the primary configuration file, `CATALINA_BASE/conf/server.xml`. This file determines how the Tomcat instance is structured in random access memory. Several important components are defined including server, services, connectors and virtual hosts. Key elements in `server.xml` that configure these components are covered as well as considerations for modifying these elements.

4.1. Role of server.xml

The `server.xml` is the primary configuration file for Tomcat. The elements in this file define all of the components of the Tomcat instance.

- A Tomcat instance, or address space, is described by one `server.xml` file.
- Multiple instances of Tomcat can run on one computer. This is possible because you can start Tomcat with a `server.xml` tailored for a specific Tomcat instance. This will be described in more detail in a future lesson.
- Exercise caution when updating this important configuration file. Tomcat startup failure can result when modifications are not properly performed. Make a backup of this file prior to changing the XML elements within this file.

4.2. Instance layout



❖ 4.2.1. Server

The server is identified in the `server.xml` by coding the `<Server>` element. The server is the container for one or more services.

❖ 4.2.2. Service

A service is identified in the `server.xml` by the presence of a `<Service>` element. The service is the container for one and only one engine-"Catalina". The engine contains one or more **virtual hosts**.

❖ 4.2.3. Virtual Host

A virtual host serves as the container for websites. The host is assigned a name; this name is the DNS (domain name service) identifier for this host machine as assigned by your Internet domain provider.

❖ 4.2.4. Context

A context represents a web application within a host. A context description in `server.xml` identifies the context path and document base for a web application. Context descriptions can also be placed in `CATALINA_BASE/conf/context.xml`.

4.3. server.xml elements

❖ 4.3.1. <Server>

This is the root element of `server.xml`. Representative attributes:

- **className** class name implementing `org.apache.catalina.Server` interface.
- **port** the port on which the server listens for the shutdown command.
- **shutdown** the shutdown command for this server.

❖ 4.3.2. <Service>

Generally one service is defined within a server. Representative attributes:

- **className** class name implementing `org.apache.catalina.Service` interface.
- **name** the display name of this service.

❖ 4.3.3. <Connector>

This element represents a component that communicates with clients external to Tomcat. Here we consider the HTTP connector element. Representative attributes:

- **acceptCount** maximum queue length for incoming requests when all threads are in use; default is 100.
- **enableLookups** boolean value indicating if DNS lookups should be performed on `request.getRemoteHost()` method calls; default is true.
- **maxPostSize** maximum number of bytes permitted on POST requests; default is 2 megabytes.
- **maxThreads** maximum number of request threads that connector is permitted to create; default is 200.

- **redirectPort** port that handles SSL communication; if a request has a matching `security-constraint` that requires SSL transport then the request is redirected to this port.
- **SSLEnabled** boolean value indicating if this connector can handle SSL requests; default is false.
- **tcpNoDelay** boolean value indicating if TCP_NODELAY will be set on server socket; default is false.

❖ 4.3.4. <Engine>

This element defines an engine, a software component that controls the request processing for a service. This element must occur once within a service element and must follow all connectors for that service. Representative attributes:

- **className** class name implementing `org.apache.catalina.Engine` interface
- **defaultHost** host that will process requests directed to host names on this server but which are not specified in this configuration file; this host name must match one of the names defined on the `Host` element.
- **jvmRoute** unique identifier used in load balancing that is appended to the session identifier so that the front end proxy (e.g. Apache) can route all requests in a session to the same Tomcat instance (server).
- **name** is the name used in log messages; must be provided and assigned a unique value if more than one Service element is defined in a Server configuration.

❖ 4.3.5. <Host>

This element defines a virtual host within the engine. The virtual host associates a network name (e.g. **localhost**) with the server that runs Catalina. Network names must be registered in the **Domain Name Services** (DNS) server that manages the Internet domain to which the server belongs. Representative attributes:

- **appBase** application base for this virtual host specified as an absolute path name or a path relative to CATALINA_BASE. This is the path to the folder where web applications are stored. The default is `webapps`.
- **className** class name implementing `org.apache.catalina.Host` interface.
- **autoDeploy** boolean that indicates if Tomcat should regularly check `CATALINA_BASE/webapps` for new or updated web applications (normally copied to the folder as WAR files) and `CATALINA_BASE/conf/Catalina/`*`hostname`* for context descriptor files (e.g. `ReplicateDemo.xml`). Default is true.

- **deployOnStartup** boolean that indicates if Tomcat automatically deploys web applications on server startup. Default is true.
- **name** network name of this virtual host as defined in the DNS server that manages your Internet domain.
- **unpackWARs** boolean indicating if WAR files placed in the application base directory should be unpacked by Tomcat; if false, then web applications are run from the compressed WAR file. Default is true.

❖ 4.3.6. <Context>

This element represents a web application within a virtual host. Context data can also be placed in `CATALINA_BASE/conf/context.xml` and will be applied to all web applications. With the exception of `server.xml` only one `Context` element can be present in a file. A `context.xml` can be stored in the `META-INF` directory of a web application. If so, then Catalina will copy this file to `CATALINA_BASE/conf/Catalina/`*`hostname/applicationContextPath`*`.xml`. Note that the file is renamed to match the application context path (if the path is multi-level then the `#` is used to indicate folder boundaries, e.g. `myWebAppBase#mySubDirectory`). Representative attributes:

- **className** class name implementing `org.apache.catalina.Context` interface.
- **cookies** boolean value that if set to true (the default) indicates that cookies will be utilized to track session identifier on the client computer. If set to false, session identifier will not be stored in cookies and the application must use another technique to track the session such as URL rewriting.
- **docBase** document base (context root) for this web application relative to the host's `appBase`. This can also be the name of the path to a WAR if you are executing a web application directly from a WAR file or the absolute path if the document base is not located under the application base of the host. Do not specify this attribute in the `META-INF/context.xml` of a web application because its value will be determined based on the path to `META-INF` from the `appBase`.
- **path** context path for the web application. This value is matched against the URL (e.g. `http://localhost:8080/`**`myWebApp`**). This attribute is only coded in `server.xml` and then only if the web application's document base is not located under the host's application base. If the document base is located beneath the application base, then the context path is the same as the document base. If `CATALINA_BASE/conf/Catalina/`*`hostname/applicationContextPath`*`.xml` is present then the context path is the same as the XML file name (excluding the trailing `.xml`).
- **reloadable** boolean value that indicates if Catalina should check for changes in `WEB-INF/classes` and `CATALINA_HOME/lib` and automatically reload the web application

if changes are detected. Default is false because of the potentially high overhead of this detection feature.

- **unpackWAR** boolean value indicating if a WAR file with a name that matches the context path should be automatically unpacked. Default is true.

❖ 4.3.7. <Realm>

A realm is a database of user names, passwords and roles. This element can be placed within `Engine`, `Host`, or `Context` elements. The attributes available vary depending on the type of realm. The attributes listed below are representative of the `DataSource` realm. For attributes associated with other types of realms, consult the Tomcat documentation.

- **dataSourceName** name of the JNDI JDBC data source. This data source must be configured in Tomcat. We will learn how to accomplish this in a later chapter.
- **roleNameCol** name of the column in the roles table that contains a role name. This role is associated with a user name in same table row (see below).
- **userCredCol** name of the column in the users table that contains the user's password (credentials).
- **userNameCol** name of the column in both the users table and the roles table that contains the user's name.
- **userRoleTable** name of the roles table created in the relational database.
- **userTable** name of the users table created in the relational database.

❖ 4.3.8. <Valve>

A valve is a Java class that will be inserted into the request processing sequence. All valves implement `org.apache.catalina.Valve`. Tomcat offers several `Valve` implementations. The `className` attribute value indicates the implementation and determines the attributes that are available to you. Here we consider the attributes available for the implementation of the **Remote Address Filter valve** (**Note:** the Remote Host Valve, `org.apache.catalina.valves.RemoteHostValve`, supports identical attributes that can be applied against a host name as opposed to an IP address):

- **className** Java class name of a Valve implementation; to use the remote address filter valve implementation specify `org.apache.catalina.valves.RemoveAddrValve`.
- **allow** comma-separated list of regular expressions against which the remote IP address will be compared. If a match is found, the request will be accepted else the request will not be accepted. If this attribute is omitted, all requests regardless of IP address will be accepted unless the IP matches a pattern listed in `deny` (see next attribute).

- **deny** comma-separated list of regular expressions against which the remote IP address will be compared. If a match is found, the request will be denied. If this attribute is omitted, all requests regardless of IP address will be accepted unless the `allow` attribute is present.

Exercise 3: Testing the Access Log

20 to 30 minutes

In this exercise you will modify the `server.xml` file to disable and enable the access log.

1. Shutdown Tomcat. Delete all the log files in `CATALINA_BASE/logs` folder.
2. Start Tomcat and and go to `http://localhost:8080`.
3. Shutdown Tomcat. In File Manager, navigate to the log directory. You will see a file whose name begins with "localhost_access_log". Open the file in a text editor. You will observe log entries for HTTP "get" requests.
4. Close your edit session for this file.Delete this file.
5. Edit `server.xml` and comment out the "Valve" entry for the "AccessLogValve". The open comment tag is `<!--` and the close comment tag is `-->`. Save the file.
6. Start Tomcat and go to `http://localhost:8080`.
7. In File Manager, navigate to the log directory. You will not see a file whose name begins with "localhost_access_log" because the valve entry in `server.xml` is commented out.
8. Shutdown Tomcat. Remove the comment tags you introduced earlier. Save the file.
9. Start Tomcat and go to `http://localhost:8080`.
10. Verify that the access log has been created by the valve.

Conclusion

In this lesson, you have learned:

- The role of server.xml.
- The components of the Tomcat memory address space.
- The elements in server.xml that define these components.
- How to change the configuration by updating server.xml.

LESSON 5
Deploying Web Applications

Topics Covered

☑ The JEE specification for web applications.

☑ The purpose of the document base and how it can be described in `context.xml`.

☑ The default context descriptor.

☑ Placing web folders in the application base.

☑ Deploying a WAR file to Tomcat.

☑ The `autoDeploy` option.

Introduction

The JEE (Java Enterprise Edition) specification dictates the folder architecture of a website and the function of servlets and JSPs. Placement of important descriptors such as `web.xml` and `context.xml` are given. The administrator must understand this architecture because the Tomcat container has been developed to expect adherence to this specification.

5.1. JEE Specification for Web Applications

❖ 5.1.1. Servlets and JSP

Servlets are the foundation of JEE web technology. All requests and responses are handled by servlets. A servlet's lifecycle is controlled by Catalina, the servlet container. When a servlet is initially loaded, the `init()` method is called by Catalina. When a servlet is about to be removed from memory, the `destroy()` method is invoked. A servlet's `service()` method is called by Catalina for a request intended for that servlet. The implementation of this method in `javax.servlet.http.HttpServlet` calls a particular method based on the type of HTTP request. For example, an HTTP GET results in a call to the `doGet` method. This method is normally overridden by a subclass of `HttpServlet` written by the developer.

❖ 5.1.2. Model View Controller (MVC) Design Pattern

MVC is a design pattern that specifies roles for servlets and JSPs. The role of a servlet is that of a **controller**. A **controller** handles requests and determines what course of action to take. The servlet interacts with the **model**. The model is represented by **JavaBeans**. A JavaBean is a Java class that has the following characteristics:

- Default, zero argument, public constructor
- Private member variables (e.g. `private String fullName`)
- Optional `getXXX` and/or `setXXX` methods (e.g. `public String getFullName`)

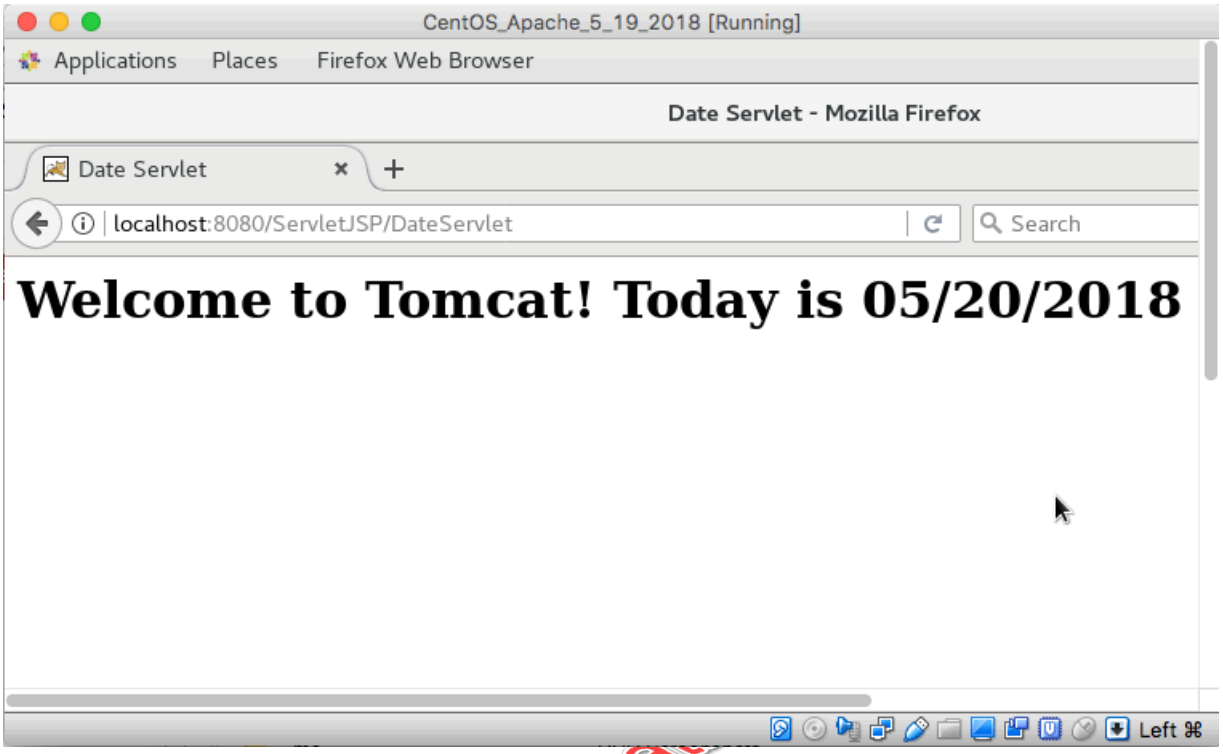
A **JavaBean** represents an entity in the business model, for example an employee. The business model data is incorporated with HTML so that a **view** of the data is sent to the client. A view is constructed by a JSP (Java Server Page). Developing HTML in a JSP is more convenient for the developer as opposed to building HTML output in a servlet.

Demo 5.1: tomcat-deploying-web-applications/Demos/DateServlet.java

```
1.  package com.webucator.servlets;
2.
3.  import jakarta.servlet.ServletException;
4.  import jakarta.servlet.annotation.WebServlet;
5.  import jakarta.servlet.http.*;
6.  import java.io.*;
7.  import java.text.*;
8.  import java.util.*;
9.
10. @WebServlet("/DateServlet")
11. public class DateServlet extends HttpServlet {
12.     private static final long serialVersionUID = 1L;
13.         public DateServlet() {
14.             super();
15.         }
16.     protected void doGet(HttpServletRequest request,
17.             HttpServletResponse response)
18.                 throws ServletException, IOException {
19.         PrintWriter pw=response.getWriter();
20.         SimpleDateFormat sdf=new SimpleDateFormat("MM/dd/yyyy");
21.         pw.println("<html>");
22.         pw.println("<head><title>Date Servlet</title></head>");
23.         pw.println("<body>");
24.         pw.println("<h1>Welcome to Tomcat! Today is " +
25.             sdf.format(new Date()) + "</h1>");
26.         pw.println("</body>");
27.         pw.println("</html>");
28.     }
29.     protected void doPost(HttpServletRequest request,
30.             HttpServletResponse response)
31.                 throws ServletException, IOException {
32.     }
33. }
```


Code Explanation

This servlet displays a welcome message that includes the current date:



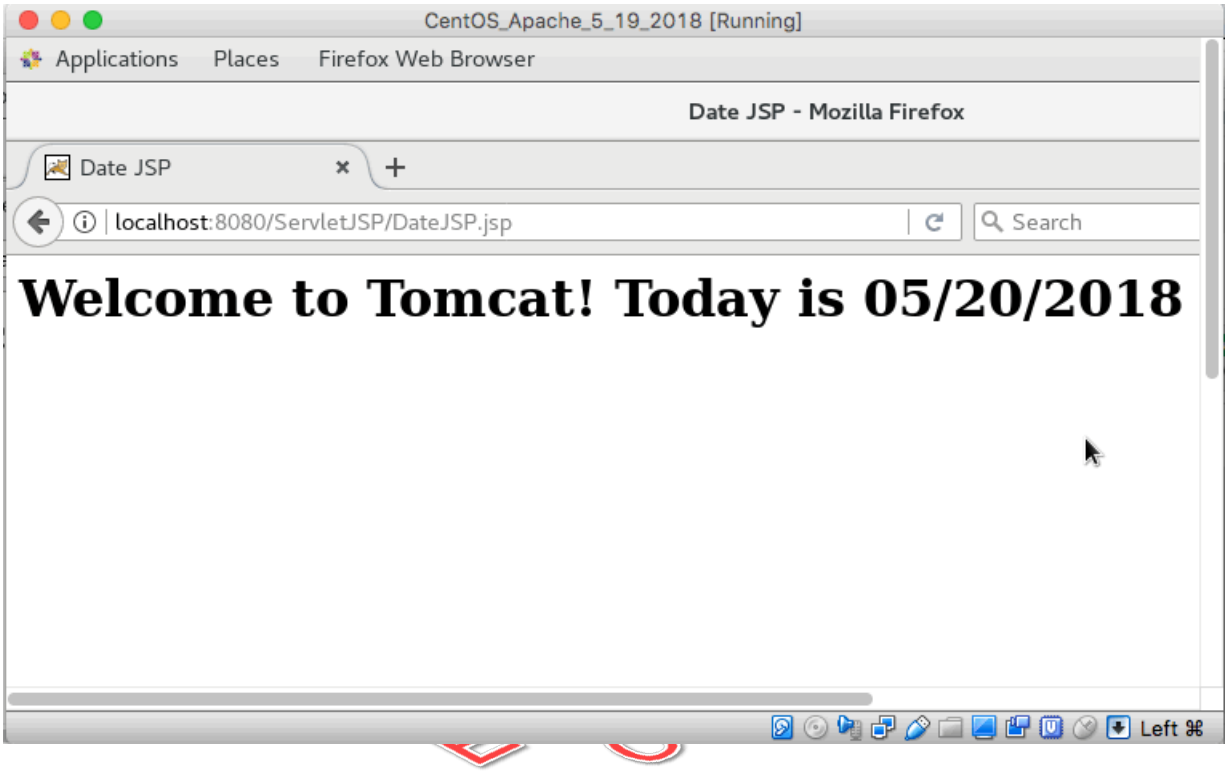
HTML is content is generated by calls to the `println` method. The servlet is geared toward procedural coding, making the servlet conducive to controller processing. Compare this program with the JSP presented in the next demo.

Demo 5.2: tomcat-deploying-web-applications/Demos/DateJSP.jsp

```
1.   <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2.    pageEncoding="ISO-8859-1"%>
3.   <%@ page import="java.text.*,java.util.*" %>
4.   <html>
5.   <head>
6.   <title>Date JSP</title>
7.   </head>
8.   <%
9.   SimpleDateFormat sdf=new SimpleDateFormat("MM/dd/yyyy");
10.  %>
11.  <body>
12.  <h1>Welcome to Tomcat! Today is <%= sdf.format(new Date()) %></h1>
13.  </body>
14.  </html>
```

Code Explanation

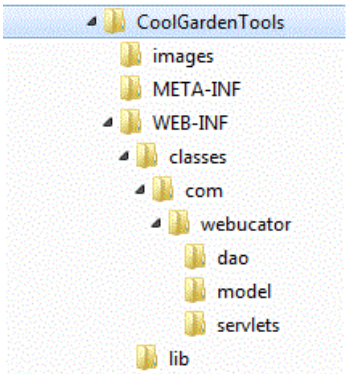
This JSP, like the servlet above, displays a welcome message that includes the current date:



The JSP source file looks very similar to an HTML document. Therefore developing a view using a JSP is more convenient-and less time consuming-than developing a view in a servlet.

❖ 5.1.3. Directory Structure

We will review in detail the directory structure of a JEE-compliant web application:



The **`WEB-INF`** folder contains the web application descriptor file, `web.xml`. This important file is discussed in more detail below. Two folders are present under `WEB-INF`:

- `classes` This directory contains the Java classes for this web application.
- `lib` This directory contains jar files for the web application.

The **`META-INF`** folder contains the optional **context descriptor**. A context descriptor is stored in a file named `context.xml`. The customary purpose of this file is to define JNDI resources, e.g. **data sources**, to the the web application.

❖ 5.1.4. web.xml

This is the primary configuration file for the web application.

Servlet mappings are stored in this file. For example, imagine the following request:

```
http://localhost:8080/SampleWebApp/FirstServlet
```

The context path is `/SampleWebApp`. The next piece of text, `/FirstServlet` is interpreted to be a URL mapping of a servlet unless a subdirectory with that name exists (a page request would contain an extension such as .html or .jsp). If the mapping is found in `web.xml` then the request is forwarded to that servlet. If the mapping is not found then a HTTP 404 status is returned.

Security constraints can be placed in this file. This topic is covered in the Security lesson.

Resource references can be stored in `web.xml`. The resource reference identifies a **JNDI** resource such as a data source.

A welcome file list can be specified in `web.xml`. A welcome file is displayed when the URL does not request a specific page and a web page exists in the context path folder that matches the name of a welcome file.

5.2. Document Base

The document base is the root directory of the web application. The document base of each web application is inferred by Tomcat for every folder appearing under `CATALINA_BASE/webapps` directory.

5.3. Context and the Document Base

A context descriptor can be stored as `CATALINA_BASE/conf/Catalina/`*`hostName`*`/`*`contextPath`*`.xml`. The *hostName* is the name of the virtual host (e.g. `localhost`). The **context path** of the web application matches the name of the descriptor file. The context path is present in the URL following the host name and port number and is preceded by a forward slash. When a URL is processed by Tomcat, the context path is inspected and the following actions are taken:

- The application base of the virtual host is read in an attempt to locate a subfolder with a name identical to the context path.
- If a subfolder with a matching name cannot be found and if `unpackWARs` is false then an attempt is made to locate a WAR file with a name identical to the context path.
- If a folder or WAR file with a name matching the context path cannot be found, then an attempt is made to locate a context file in `CATALINA_BASE/conf/Catalina/`*`hostName`* with a name matching the context path appearing in the URL.
- If a match still cannot be found, then an HTTP status of 404 is returned to the client.

If the document base of the web application is not under `webapps`, the `docBase` attribute of the `Context` element must be present. This attribute indicates the document base of the web application. Note that the context path of the web application is identical to the name of the XML file. Accordingly, the `path` attribute does not need to be present (if the attribute is present, its value must match the context path implied by the descriptor's file name).

5.4. Default Context Descriptor

Tomcat creates a context descriptor for a web application using default values if a context file is not present for that web application.

5.5. Placing the Web Application Folders and Files under the Application Base

You can manually create a web application folder and its associated directories (i.e. `WEB-INF`) under `webapps`. Alternatively, you can create the directory structure in a separate location and copy the structure to `webapps`.

The drawback of placing all web applications under `webapps` is that all disk I/O is concentrated on this one folder. Placing a context in another directory or on another drive may contribute to better performance. A context descriptor (an XML file with one `Context` element) can be deployed to `CATALINA_BASE/conf/Catalina/HostName` . This descriptor can specify a document base under a directory different from `webapps`. To implement this option, you create the directory structure for the website and reference that location in the context descriptor. This option will be explored in one of the lesson exercises.

Deployment using a **web archive** (WAR) file is standard operating procedure in Tomcat. A developer can build the web application in an **IDE** (Integrated Development Environment) and then create a WAR file and copy this file to the `webapps` directory.

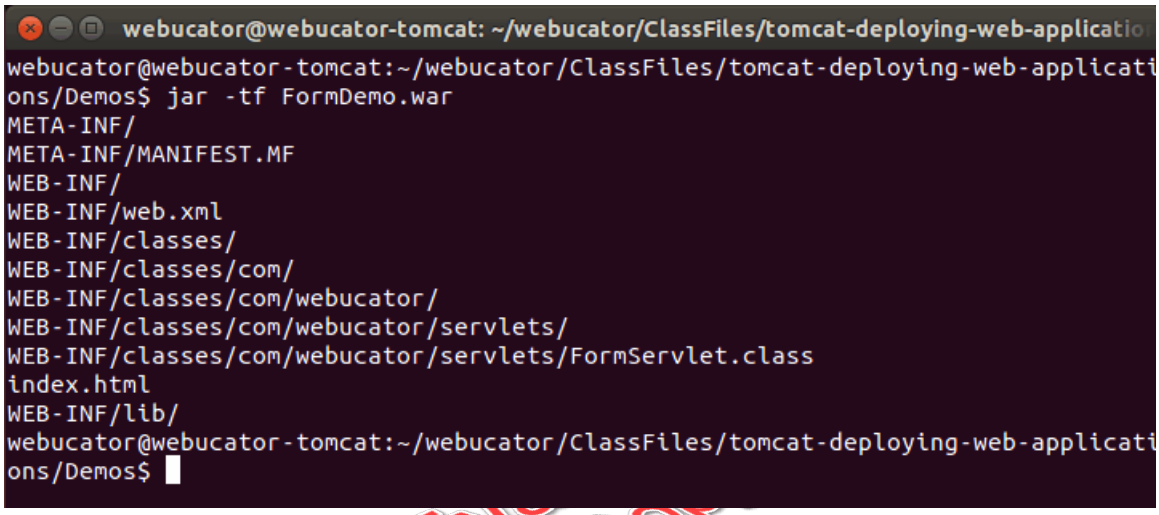
5.6. Deploying a WAR file

A WAR file is a compressed ("zip") file containing the context root folder and subfolders of the web application. The WAR file can be created in several ways. For example, you can use the `jar` utility located in `JAVA_HOME/jdk/bin`. Alternatively, you can use an `Ant` task. Ant offers a `war` element that facilitates the creation of a web archive file.

When a WAR file is placed under `webapps`, the file is uncompressed automatically by Tomcat if the `Host` element specifies `unpackWARs=true`. The web application will be deployed if the `Host` element specifies `autoDeploy=true`.

The demo directory of this lesson contains `FormDemo.war`:

- This file is a **web archive** (WAR) of the Form Demo web application.
- The file can be placed in the `CATALINA_BASE/webapps` directory. If `unpackWARs` is true (as specified in the `Host` element in `server.xml`) then the WAR file will be exploded and an uncompressed folder with the same name will be created.
- If `autoDeploy` is true then the web application will be ready to receive requests.
- To view the contents of the web archive file, start a command prompt and navigate to the directory containing the WAR file.
- On the command line type `jar -tf FormDemo.war`. You should now see the contents of the WAR file.



5.7. AutoDeploy

The autodeploy (automatic deployment) option can be activated for a virtual host. With automatic deployment in place, WAR files copied into `webapps` are started so that requests can be processed by the new or updated web application. In addition, context files placed in `CATALINA_BASE/conf/Catalina/`*`hostName`* will cause the corresponding web application to be started.

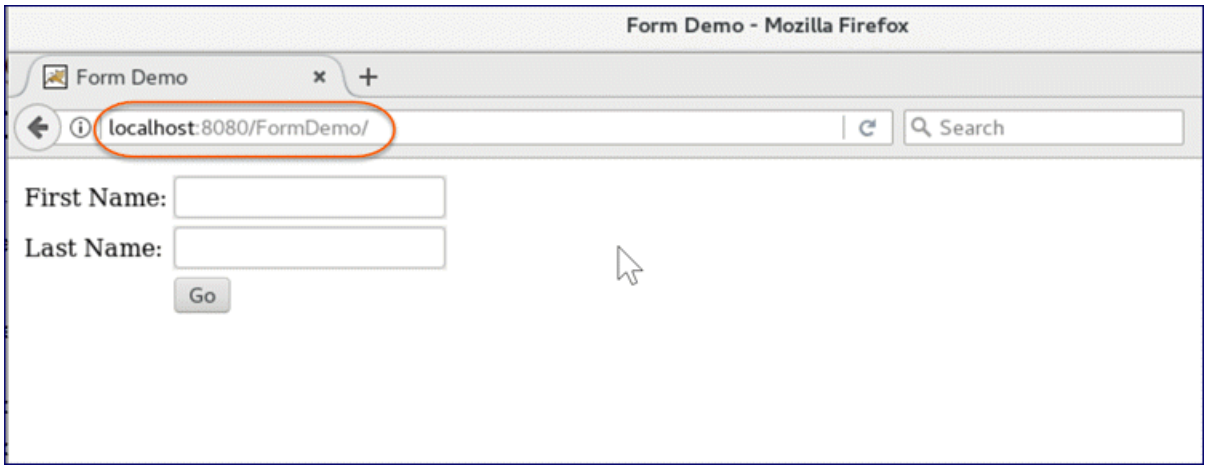
Exercise 4: Deploying a Web Application to Tomcat Using a WAR file

10 to 15 minutes

In this exercise, you will copy the WAR file provided in the demo to the `/webapps` directory under CATALINA_HOME.

1. Copy the WAR file `FormDemo.war` provided in the `Exercises` folder to `CATALINA_BASE/webapps`.
2. Check the console log to ensure that Tomcat detects the WAR file.
3. View the webapps folder in File Manager to verify the WAR file has been exploded.
4. Test the web application in your browser by entering `http://localhost:8080/FormDemo`.

Solution



Exercise 5: Deploying a Web Application to Tomcat Using a Context Descriptor File

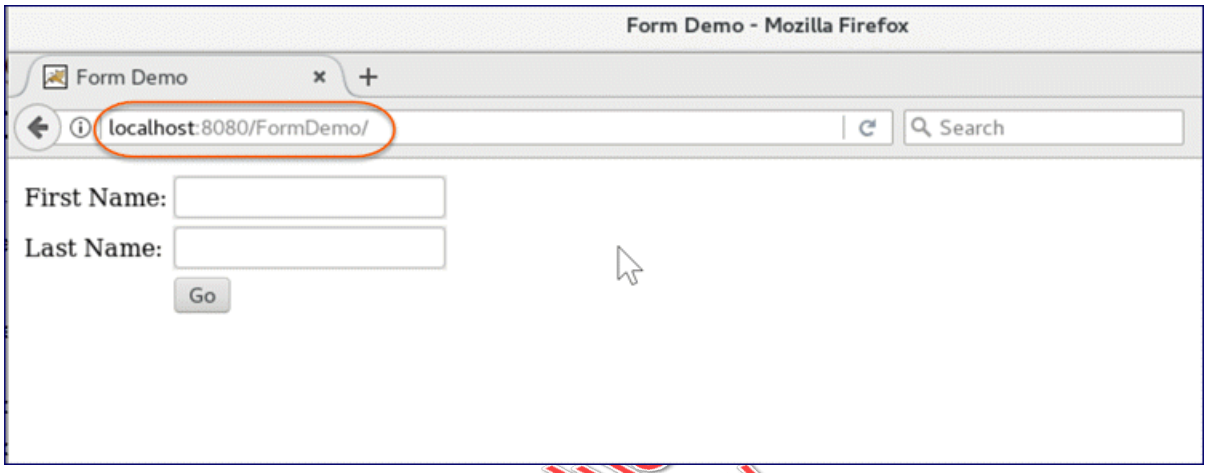
30 to 45 minutes

In this exercise, you will build the directory structure of a web application and deploy the application using a context descriptor file. The web application is **Form Demo** and will be accessible through its context path, `/FormDemo`.

1. Create a folder under the `root` folder named `ContextXMLDeployment`.
2. Create a subfolder named `FormDemo`. This folder, `FormDemo`, is the context root of the website (**Note:** we are giving the context root folder the same name as the context path, including upper case characters where appropriate. This is not technically required because the `docBase` attribute can specify an arbitrarily named directory. However, using a directory name that is identical to the context path is recommended for clarity).
3. Create the folder(s) under `FormDemo` that conform to the JEE specification (refer to the review of the directory structure of a JEE-compliant website presented earlier in this lesson).
4. A web descriptor file, `web.xml`, is provided in the `Exercises` folder. Copy this file to the correct folder.
5. A servlet class file, `FormServlet.class` is provided in the `Exercises` folder. Create the package folder hierarchy for this servlet given that the package is `com.webucator.servlets` (*Hint:* each component of the package name maps to a folder, starting with `com`). The folder hierarchy must be placed in the correct location.
6. An HTML file named `index.html` is provided in the `Exercises` folder. Copy this file under the `FormDemo` directory.
7. A context descriptor file, `FormDemo.xml`, is provided in the `Exercises` folder. Open this file in your text editor.
8. Make note of the `path` attribute value. This is the context path of the website. Locate the `docBase` attribute. Change the value to `/ContextXMLDeployment/FormDemo`. Save your changes.
9. Stop the Tomcat server. Undeploy the `FormDemo` web application by deleting the `FormDemo` directory under `webapps`. Delete `FormDemo.war` as well.
10. Copy `FormDemo.xml` to `CATALINA_BASE/conf/Catalina/localhost/`.

11. Start Tomcat. Test the web application in your browser by entering `http://localhost:8080/FormDemo`.

Solution



Conclusion

In this lesson, you have learned:

- The JEE specification for web applications.
- The purpose of the application base and how it can be described in context.xml.
- The default context descriptor.
- The location of web folders in the application base.
- How to deploy a WAR file to Tomcat.
- The function of the AutoDeploy option.

LESSON 6
The Tomcat Manager

Topics Covered

☑ The purpose of the Tomcat manager application.

☑ Deploying, reloading, starting, and stopping web applications.

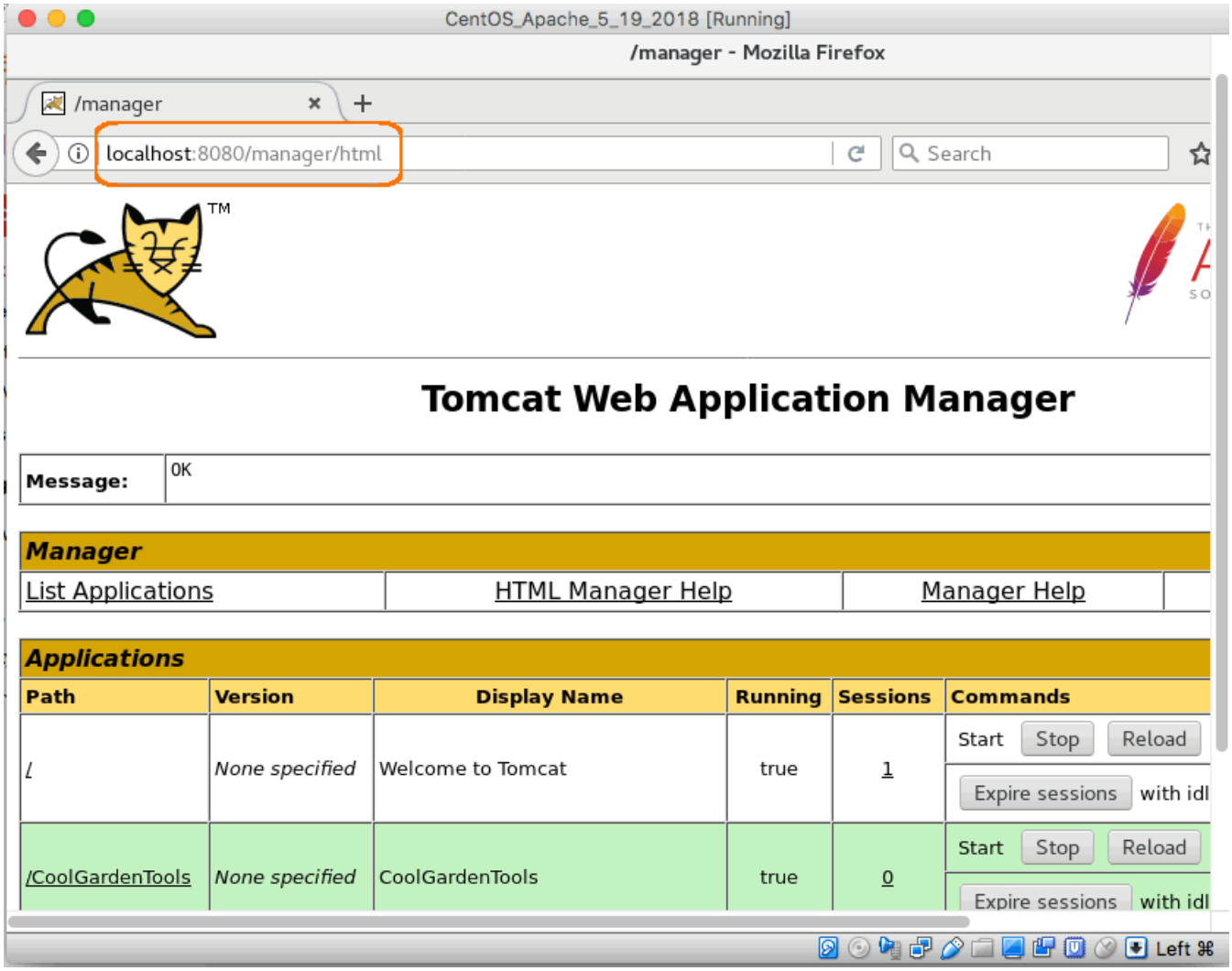
☑ Listing server status.

Introduction

The manager web application shipped with Tomcat provides a convenient tool to manage web applications. In this lesson, you will learn how to deploy, reload, start and stop web applications.

6.1. /manager Web Application

The manager application is a useful tool for managing web applications. The address is `http://localhost:8080/manager/html`. User name and password must be provided (explained below). Then the web page will be displayed:



You must set up a user account and a role of "manager-gui" in `CATALINA_BASE/conf/tomcat-users.xml`. Edit this file and add the following lines at the bottom of the file:

```
<role rolename="manager-gui"/>

<user username="myUserName" password="myPassword" roles="manager-gui"/>
```

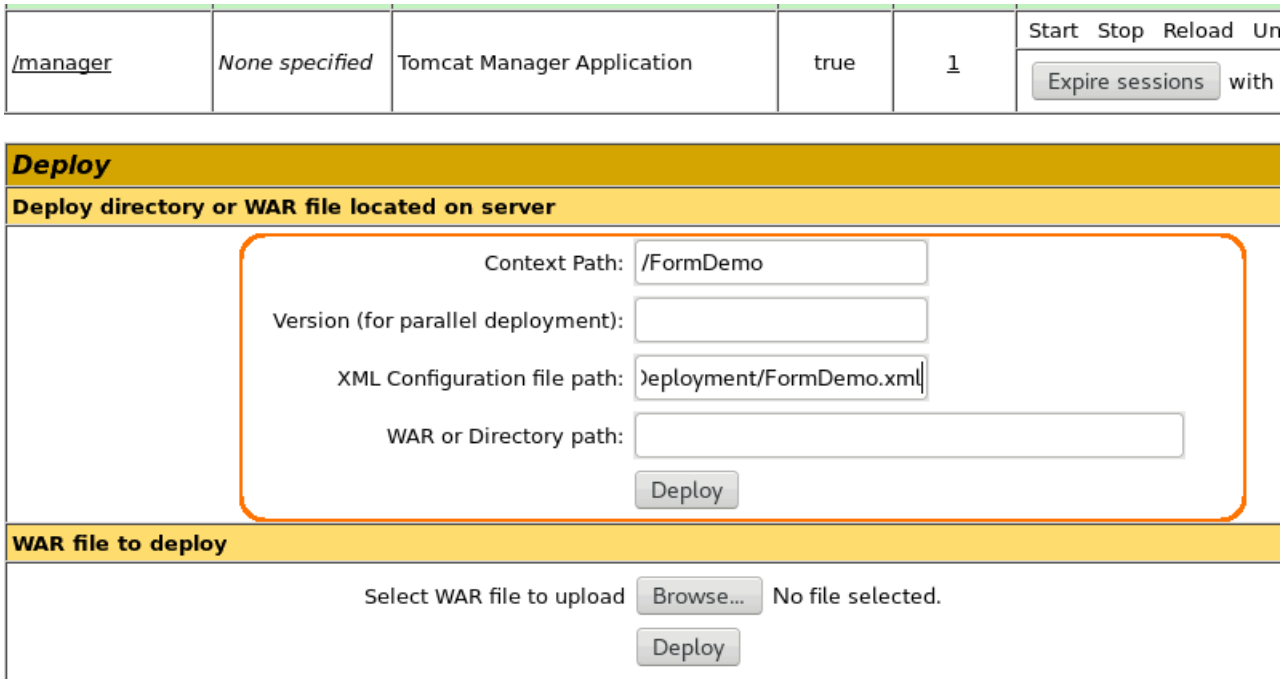
Replace *myUserName* and *myPassword* with a username and password of your choice. Save the file.

Note on text editing in Virtual Box: To enter a quotation mark or an apostrophe in gedit, press quotation mark or apostrophe and at the same time press right **option** on the Mac or press right **Alt** on Windows.

6.2. Managing Web Applications

❖ 6.2.1. Deploying

One of the common tasks an administrator performs is deploying Web applications. The deploy option of the manager application is shown below:



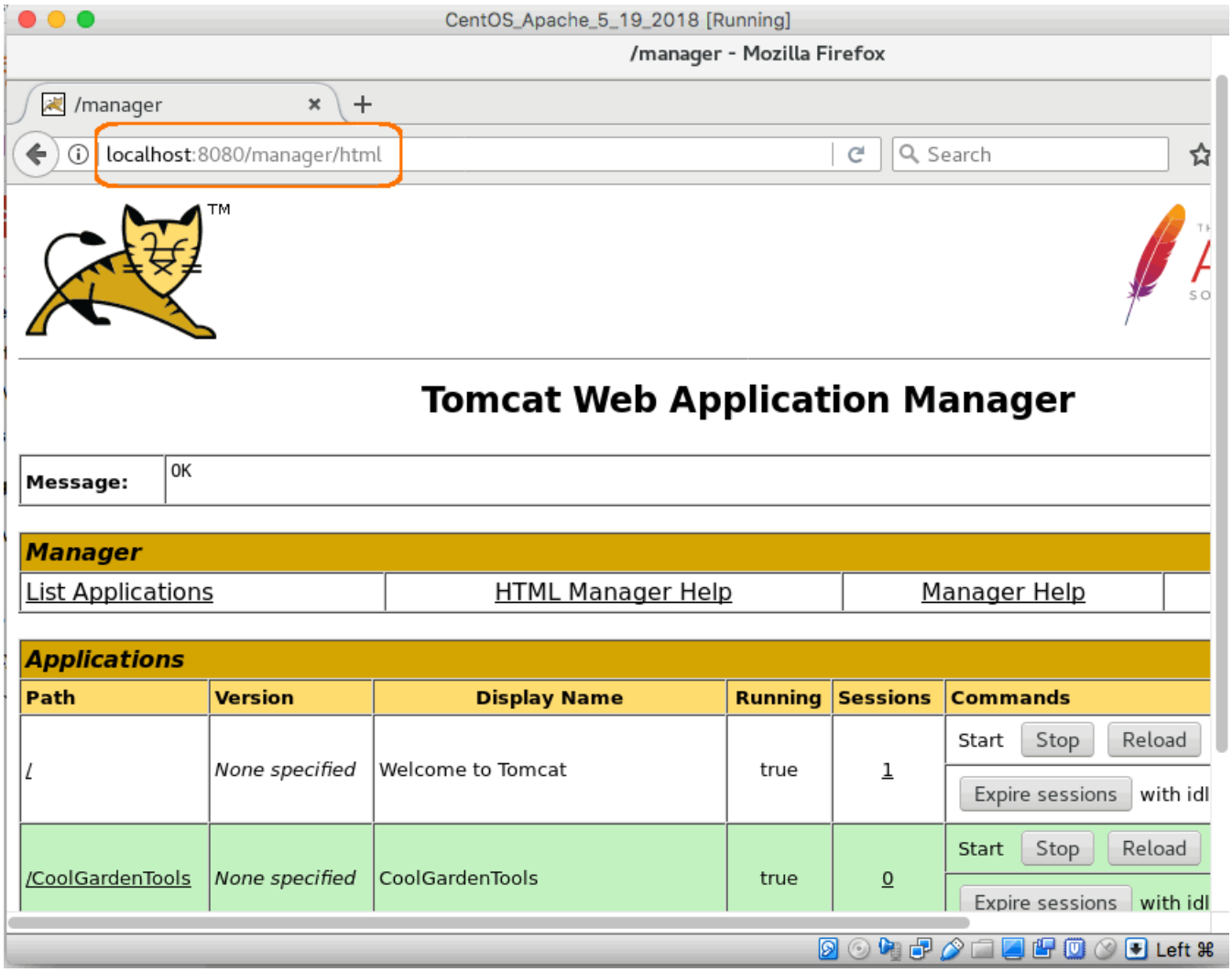
You can deploy a context descriptor XML file or a WAR file. For WAR files, a **Browse** button is available if you want to navigate the file system in locate the WAR file.

In the screen print above, I am deploying a web application using a context descriptor XML file. I have provided the context path of the web application. The first character should be a forward slash ("/"), e.g., `/FormDemo`. The context path is the means of identifying an incoming request with a website. The context path must be unique for this Tomcat server.

Upon clicking the **Deploy** button, the manager application will copy the context configuration file to `CATALINA_BASE/conf/Catalina/localhost`. The file must specify a `docBase` attribute indicating the document base of the web application (**Note:** this document base does not have to be present under `webapps` and generally is located in a directory separate from `webapps`).

❖ 6.2.2. Listing Deployed Applications

When you start the manager application, a list of deployed Web applications is displayed:



Several options are available on this list. We will look at them now.

❖ 6.2.3. Reload Existing Applications

Select this option if you want to reload Java classes from `/WEB-INF/classes` or jar files from `CATALINA_BASE/lib`. Note: the reload is not supported if the web application was not unpacked from a WAR file.

❖ 6.2.4. Starting/Stopping

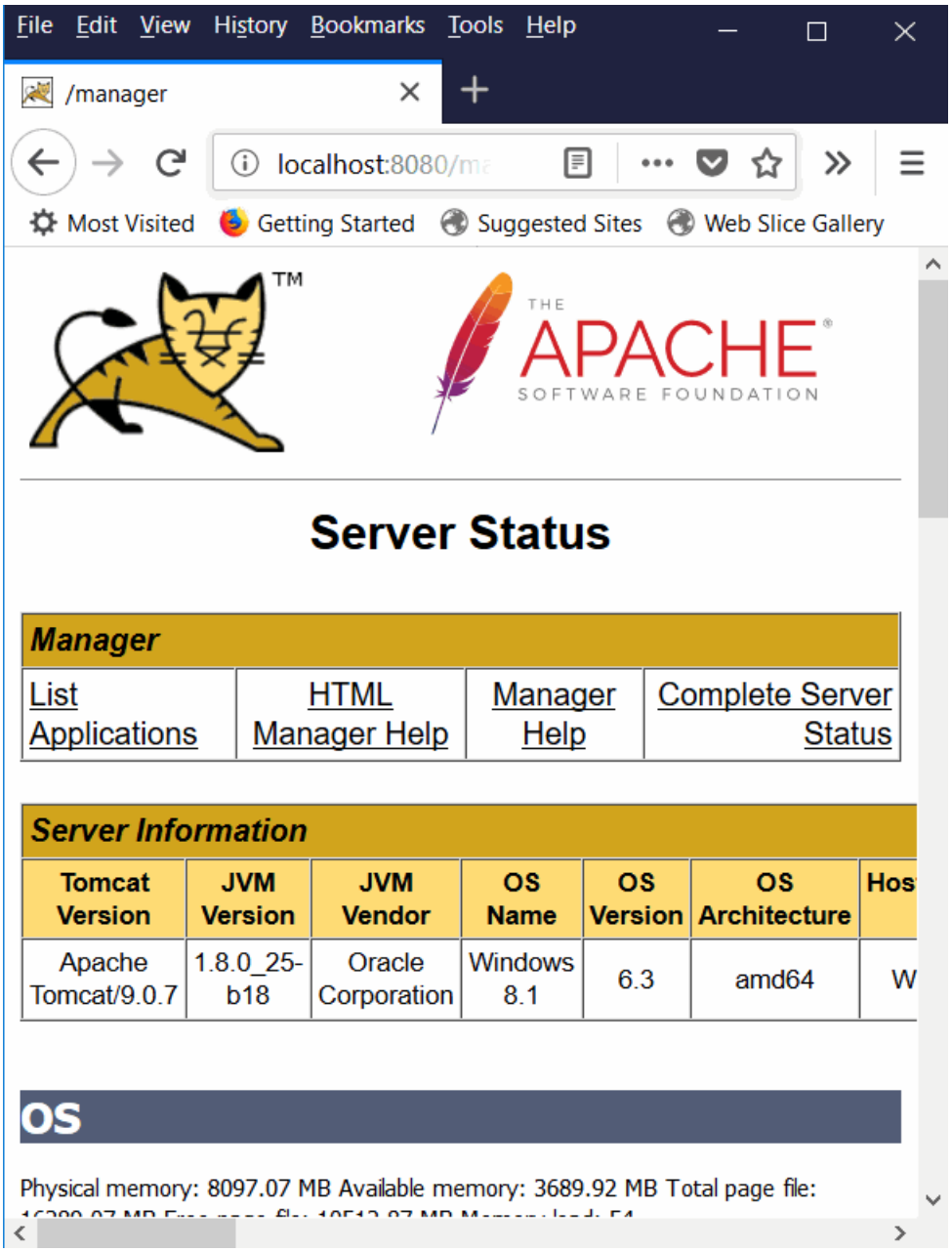
A Web application is started automatically when it is deployed. To stop the Web application, select the stop button on the appropriate application. A stopped application will return an HTTP status of 404 for any URL request made for that application. To start the Web application, select the start button.

❖ 6.2.5. Undeploying

Select this option if you want to undeploy (remove) the Web application. Use caution because the document base and subdirectories will be deleted in addition to the corresponding WAR if it exists.

6.3. Listing Server Status

The **server status** link can be selected to provide JVM data such as memory utilization and thread count. The JVM version is also available.



6.4. Listing Security Roles in the User Database

To list the security roles stored in the user database, go to `http://localhost:8080/manager/roles`. A user database must be installed and the `manager.xml` file must be updated to include a `ResourceLink` to this database. The user database is defined in `server.xml` in a `Resource` element.

Exercise 6: Administration Tasks Using Manager

20 to 30 minutes

In this exercise, you will perform several administrative tasks using the Tomcat manager. All tasks are to be performed using the manager web application.

1. The Form Demo web application was deployed in an earlier lab. Undeploy this web application using the manager application.
2. Deploy the Form Demo web application using the WAR file provided in the `Exercises` folder.
3. Test the web application by clicking the link on the manager page.
4. Return to the manager by hitting the **back** arrow on your browser. Stop the web application. Try to access the web application. What happens?
5. Start the web application. Retry the request by clicking the link on the manager page. The website should be operational again.
6. Undeploy the web application. Check the `webapps` folder to verify the WAR file and context directory for the Form Demo web application have been removed.
7. Refer to the `Exercises` folder and locate `FormDemo.xml`. This is the context descriptor file for the "FormDemo" application and contains a `docBase` reference to the directory `/ContextXMLDeployment/FormDemo` that you created in an earlier lab. Deploy the web application from the manager using this file.
8. Verify the web application is operational by pointing your browser to `http://localhost:8080/FormDemo`.

Solution

To undeploy the web application, click the `Undeploy` button in box bordering the `Form Demo` web application entry.Click `OK` to proceed with the undeploy operation.

To deploy the Form Demo web application, provide the context XML file name and then click the `Deploy` button. The manager application will indicate that the XML file has been successfully deployed.

Conclusion

In this lesson, you have learned:

- The purpose of the Tomcat manager application.
- How to deploy, reload, start and stop web applications.
- How to list server status.

LESSON 7
JNDI Data Sources and JDBC

Topics Covered

☑ The purpose of JNDI.

☑ Accessing relational data with JDBC.

☑ Database connection pooling.

☑ Configuring data sources in `server.xml` and `context.xml`.

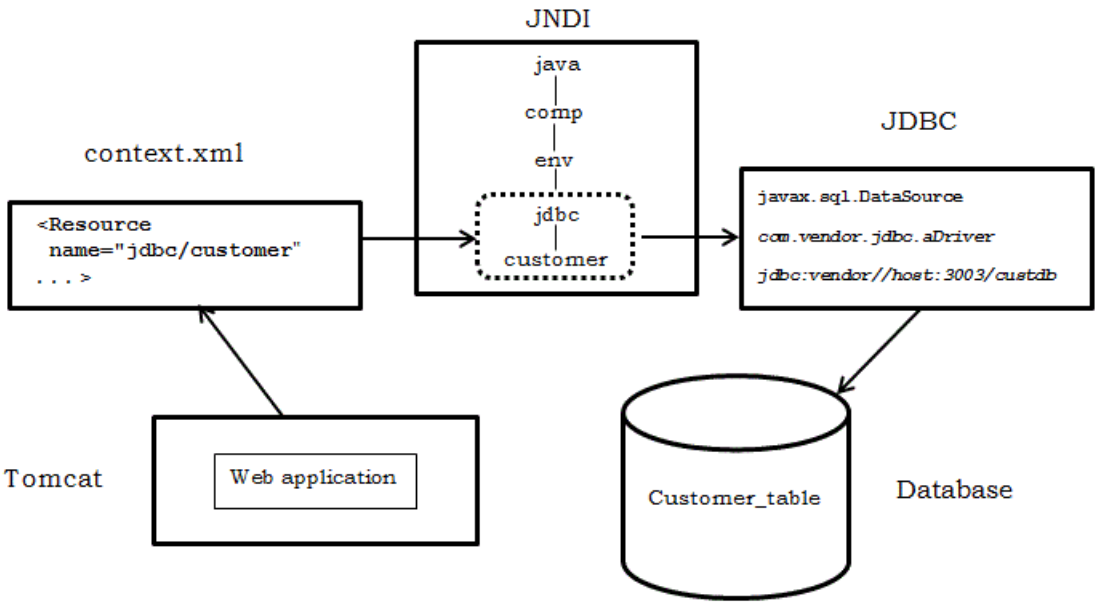
☑ Troubleshooting common data source problems.

Introduction

JNDI is an API that facilitates look-up of Java EE resources such as data sources that connect to relational databases. In this lesson you will learn how to configure JNDI data sources. The data sources represent JDBC connections to relational databases. JDBC 2.0 and later versions support database connection pooling.

7.1. JNDI

JNDI (Java Naming and Directory Interface) is a look-up facility for Java EE resources such as message queues, enterprise java beans (EJBs) and JDBC data sources. JDBC data sources are generally a vital component of web applications. Business model data is stored in relational databases such as Oracle, DB2 and MySQL. JNDI technology simplifies the application code in the model layer by removing dependencies on database URL and driver location. The components that an application requires (such as a data source) are bound to a level in the JNDI hierarchical structure. A level is referred to as a **context**.



The diagram above portrays how a Web application accesses a JDBC database as a JNDI data source. The context file defines a JNDI name. A Java class in the Web application looks up the JNDI name and it must match the name defined in the context file.

JNDI utilizes the services of JDBC to access the database. A connection object is obtained from the connection pool and returned to the application. The application can then issue SQL statements using this connection.

The hierarchical nature of JNDI is evident in the diagram. The naming convention is based on a `java` URL scheme. The root context is `comp` ("component"). The `comp` context may contain two bindings: `env` and `UserTransaction`. The `UserTransaction` is bound to `javax.transaction.UserTransaction` and is utilized in transaction processing (e.g. to issue a commit). The `env` ("environment") is bound to a subtree that is utilized for an application's environment-related bindings.

For example, a subtree might have the name "jdbc" indicating the bindings in this tree relate to JDBC data sources. This is a JEE recommendation but not a requirement (**Note:** another recommendation is `ejb` for Enterprise JavaBeans). The actual name (e.g. `customer`) of the environment data can be given a name that is meaningful to the application.

7.2. JDBC

JDBC (Java Database Connectivity) is the technology employed by Java applications to access relational databases. JDBC is available in the Java SE.

Using JDBC, a SQL statement object is created by calling the appropriate method on the `java.sql.Connection` object. For example, a prepared statement can be created by calling the `prepareStatement` method. A connection can be obtained by calling the `getConnection` method defined in `javax.sql.DataSource`. The data source is obtained by a JNDI lookup.

Three types of SQL statements can be constructed by the application using implementations of the following interfaces:

- **`java.sql.Statement`** a static SQL statement
- **`java.sql.PreparedStatement`** a precompiled ("prepared") SQL statement that can be executed repeatedly without recompilation. Placeholders (represented by the question mark, ?) can be placed within the statement (e.g. in the `WHERE` clause). The placeholders can be replaced with data values using `setXXX` methods. This statement type is recommended because of its potential performance benefit.
- **`java.sql.CallableStatement`** a stored procedure call.

❖ 7.2.1. Drivers

The ODBC (Open Database Connectivity) paradigm proposed by Microsoft in 1990 introduced the concept of **database driver**. A driver is a component that connects application code to the database manager. JDBC, founded on the same principles as ODBC, requires that the database vendor supply a driver. The driver is typically supplied in a jar file.

❖ 7.2.2. Data Sources in JDBC 2.0 and Later

JDBC 2.0 and later versions support **connection pooling**. Prior to JDBC 2.0, a Java program utilized the services of `java.sql.DriverManager` to manage the connection to the database. The Java code was responsible for providing vendor-specific database URLs. Now, starting with JDBC 2.0, the application program does not manage the connection ensuring improved utilization of database resources and minimizing connection-related code in the Java program.

Closing a connection obtained using `DriverManager` also closes `Statement` and `ResultSet` objects. Closing a connection derived from connection pooling does not close `Statement` or `ResultSet` objects.

❖ 7.2.3. Connection Pooling

Connection pooling is a technology that removes the work of maintaining a database connection from the application program. In general, this promotes superior throughput to the database manager because a pool of threads is reused, as opposed to individual Java programs starting and removing threads in an unsynchronized manner.

7.3. Commons Database Connection Pooling

Commons DBCP is the connection pooling technology utilized by Tomcat.

❖ 7.3.1. Installation

The jar file implementing DBCP is located in `CATALINA_HOME/lib/tomcat-dbcp.jar`. Since this file is already located in the `lib` directory, DBCP is installed and ready for use.

❖ 7.3.2. Guarding against Application Program Failure

Developers should be informed that connections should be closed in the Java code. This code should be placed in a `finally` block that will be executed even if the application program fails. Closing the connection releases it so that the connection can be used by another request.

❖ 7.3.3. Configuration

JNDI is utilized in conjunction with JDBC. To configure a JNDI data source, code a `Resource` element in a context file. The scope of the configuration is dependent on the location of the context file. The next section goes into more detail on the choice of location for the data source definition.

7.4. Data Source Definition

❖ 7.4.1. The Resource Element

A data source is defined to Tomcat using the `Resource` element. Here is an example:

Demo 7.1: tomcat-jndi-data-sources-and-jdbc/Demos/ResourceElement.html

```
1.    <Resource name="jdbc/gardentools" auth="Container" type="javax.sql.DataSource"
2.       maxTotal="50" maxIdle="10" maxWaitMillis="5000"
3.       username="root" password="rootpw" driverClassName="com.mysql.jdbc.Driver"
4.       url="jdbc:mysql://localhost:3306/test"/>
```

Code Explanation

Code explanation

The attributes are defined below

- **name** JNDI name of the data source relative to the `java:comp/env` context.
- **auth** specifies if the container signs on to resource manager or if the application code is responsible for signing on to the resource manager.
- **type** type of resource.
- **maxTotal** maximum active database pool connections; set this value to -1 for no limit.
- **maxIdle** maximum number of idle connections to keep in the pool; set this value to -1 for no limit.
- **maxWaitMillis** maximum time to wait for a connection in milliseconds (e.g., a value of 5000 equals 5 seconds); set this value to -1 to wait indefinitely.
- **username** user name to provide to the database server for authorization.
- **password** password for user name.
- **driverClassName** Java class name of the driver.
- **url** vendor specific uniform resource locator; this includes the host name and port number on which the database server listens for SQL requests

❖ 7.4.2. context.xml in /conf

The `Resource` element can be placed in `CATALINA_BASE/conf/context.xml`. The data source defined by the `Resource` element will then be available to all web applications.

❖ 7.4.3. GlobalNamingResources in server.xml

The resource definition can also be defined in the scope of the `GlobalNamingResources` element in `CATALINA_BASE/conf/server.xml`. The data source will be globally accessible.

❖ 7.4.4. context.xml in the META-INF Directory of the Web Application

If you want to limit access to the data source to a particular web application, then you can define your data source in `context.xml` in the `META-INF` directory of the web application. An example is presented below.

Demo 7.2: tomcat-jndi-data-sources-and-jdbc/Demos/context.xml

```
1.   <?xml version="1.0" encoding="UTF-8"?>
2.   <Context>
3.   <Resource name="jdbc/gardentools" auth="Container" type="javax.sql.DataSource"
4.      maxTotal="100" maxIdle="30" maxWaitMillis="10000"
5.      username="root" password="rootpw" driverClassName="com.mysql.jdbc.Driver"
6.      url="jdbc:mysql://localhost:3306/test"/>
7.
8.   </Context>
```

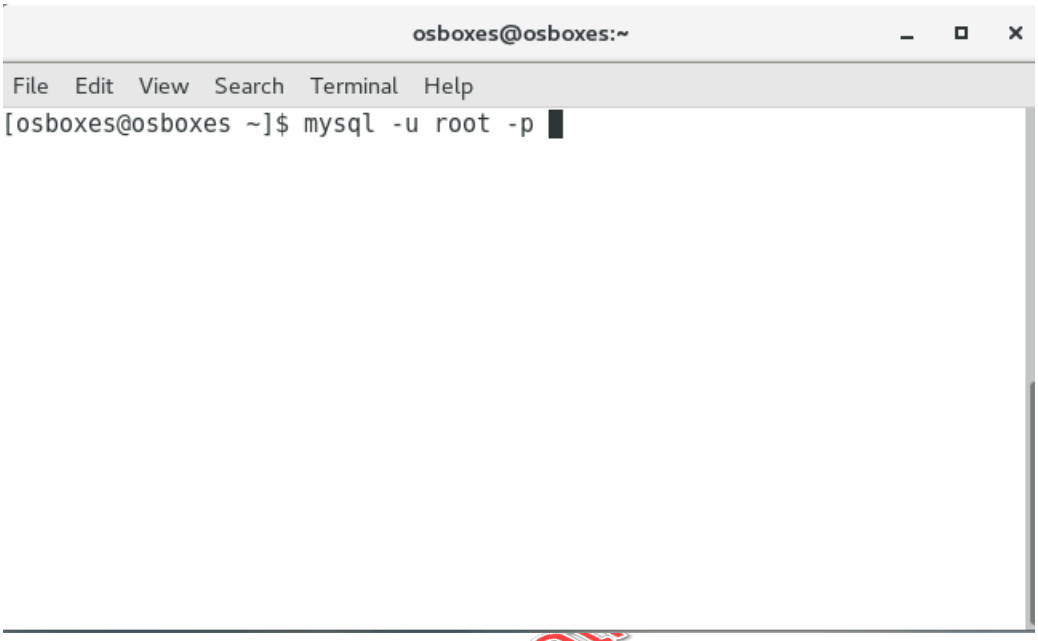

Code Explanation

Note that this file has the same name as the file in the `CATALINA_BASE/conf` directory but is present in the `META-INF` directory of the web application. Therefore, the resource applies only to the current web context.

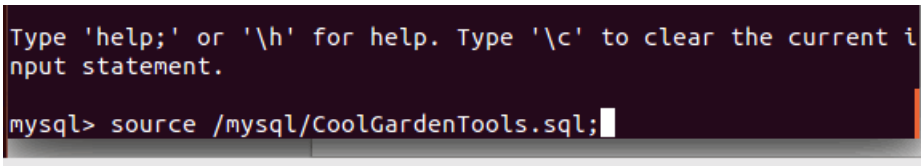
❖ 7.4.5. Creating a Data Source in MySQL

For the class room e-commerce web application we will create a data source using MySQL:

1. Start the MySQL service in a terminal window. Then execute the MySQL command line tool:



2. When prompted for the password, enter `NewPassword` and then press **Enter**.
3. Create the test database by typing `create database testExercise;`. (Note: the "test" database has been provided as the solution.) Hit **Enter**.
4. The SQL script to build the garden tools table is located in `CoolGardenTools.sql` in the `Demos` directory for this lesson. You will run this script in the mysql command line client. Enter `use testExercise` to connect to the test database and then press **Enter**. Type in `source /path/Demos/CoolGardenTools.sql;` where *`path`* is the path to the Demos directory for this lesson. The emphasized text should be replaced with the path to the class files on your computer:



Hit **Enter** to run this script. Sample output is shown below.

```
mysql> source /mysql/CoolGardenTools.sql;
Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.05 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

mysql>
```


You have now created the table that will be accessed by the e-commerce web application.

7.5. Troubleshooting

If the connection to a data source is not successful in a Java program, use the following checklist to resolve the problem:

- Ensure the database manager is started.
- Check the JNDI name referenced in the application and ensure the name matches the data source definition in a context file or `GlobalNamingResources`. JNDI will throw a "Name not found" exception in a Java program if the JNDI name doesn't match the value provided in the descriptor file.
- Check the database URL as specified in the descriptor file to ensure it's correctly coded.
- Make sure the correct driver is copied under Tomcat's `lib` folder.

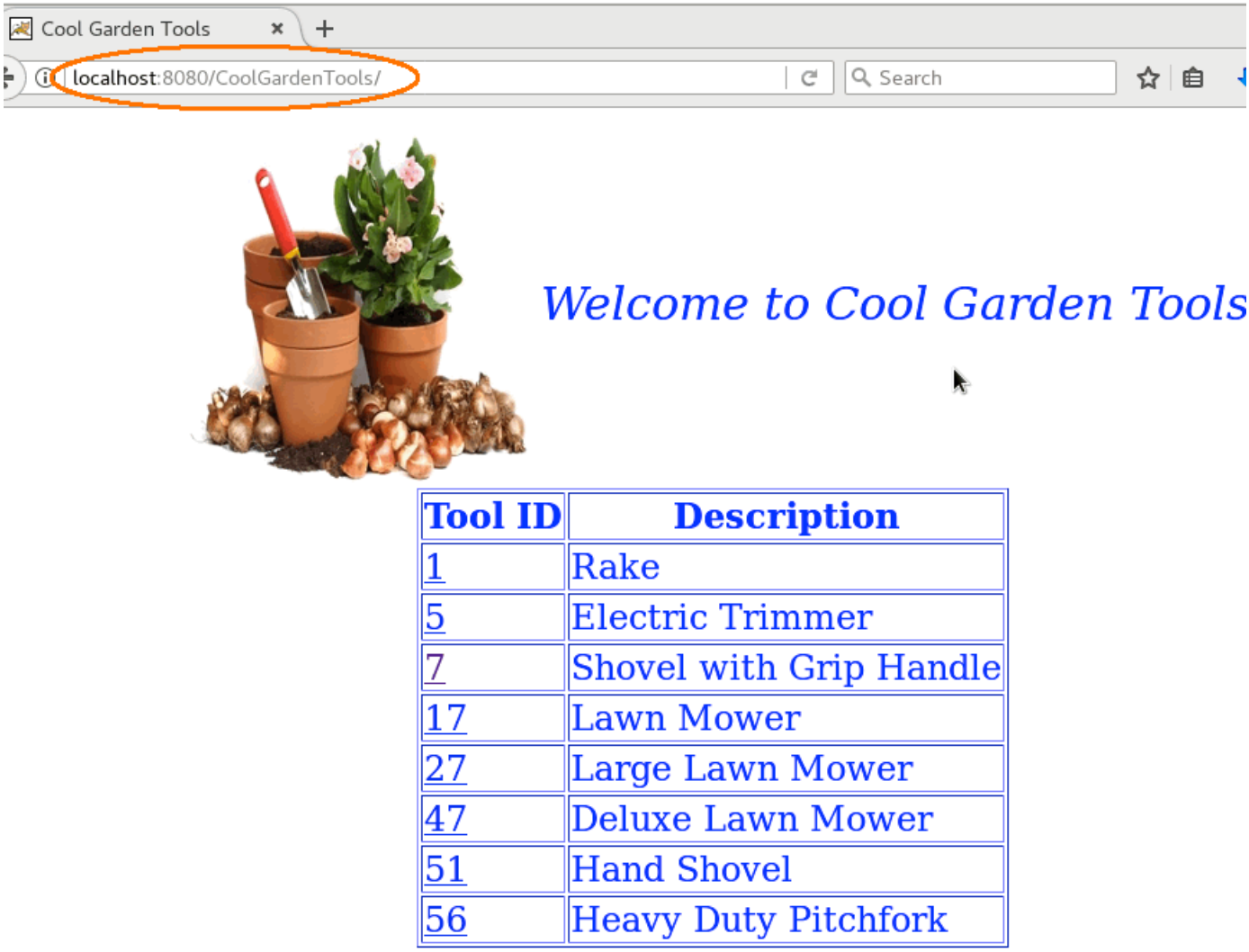
Exercise 7: Defining and Testing a JNDI Data Source

25 to 40 minutes

In this exercise, you will define the Cool Garden Tools data source to Tomcat in the `context.xml` located under the `CATALINA_HOME/conf` directory.

1. Shut down the Tomcat server.
2. The MySQL driver file, `mysql-connector-java-5.1.15.jar` is provided in the `Exercises` folder. Copy the driver to `CATALINA_HOME/lib`.
3. In a text editor, open `context.xml` in the `CATALINA_BASE/conf` folder. Using `Context.html` in the Demos folder of the current lesson as a guide, add a `Resource` element defining the Cool Garden Tools data source. Be sure to change the database name from "test" to "testExercise".
4. Save the file.
5. Deploy the Cool Garden Tools web application using the WAR file `Exercises/CoolGardenTools.war`.
6. Start Tomcat and point your browser to the Cool Garden Tools website. The tools list should be displayed. The data source is accessible to any web application in your Tomcat server.
7. Now you will deploy the data source as a web application resource, i.e, a resource that can be accessed only by Cool Garden Tools and by no other web application.
8. Stop the Tomcat server.
9. Copy `CATALINA_BASE/conf/context.xml` to the Cool Garden Tools web root folder's meta information directory, i.e., `CATALINA_BASE/webapps/CoolGardenTools/META-INF`.
10. Open `CATALINA_BASE/webapps/CoolGardenTools/META-INF/context.xml` for edit. Remove the "WatchedResource" element from the file and associated comments. Save your changes.
11. Open `CATALINA_BASE/conf/context.xml` for edit. Comment out the resource definition you added in a previous step. Save you changes.
12. Start the Tomcat server.
13. Test the Cool Garden Tools web application to verify you can still connect to the database.

Solution



Conclusion

In this lesson, you have learned:

- About the purpose of JNDI.
- How JDBC is utilized to access relational data.
- About database connection pooling.
- How to configure data sources in server.xml and context.xml.
- How to troubleshoot common data source problems.

LESSON 8
Security

Topics Covered

☑ Security considerations for web applications.

☑ The Java `SecurityManager`.

☑ Implementing Secure Socket Layer.

☑ DataSourceRealm for Web application security.

☑ The purpose of `tomcat-users.xml`.

Introduction

Security is vital to the successful operation of Tomcat. In this lesson you will learn how to secure Tomcat and the web applications associated with the server.

8.1. Web Application Security

Web applications are usually secured using elements in `web.xml`. The goal is to ensure that only authenticated users can access the website. A user is authenticated by presenting a dialog box or form requesting the username and password.

Authorization is the task of determining if an authenticated user can perform the requested task. Tasks are logically grouped in roles. Users are then assigned to these roles. For example, Nancy is a manager and is authorized to update and delete employee records. Bill is a data entry clerk and is authorized only to add an employee record to the database. Therefore we can define two roles, `manager` and `clerk`. We would assign Nancy to the first role and Bill to the second role. Additional persons might be assigned to these roles as well.

Tomcat has the capacity for performing authentication and authorization using a realm. A realm is a database of user names and passwords and is used in conjunction with `<security-constraint>` and `<login-config>` elements in the `web.xml` file.

A security constraint associates one or more roles with a URL. For example, in the manager application the URL pattern `/html/*` is associated with *manager* and *manager-gui* roles. A login configuration defines the protocol for authentication, in this case it is form-based.

User and role data is accessed by means of a **Realm**. A realm represents a database that stores authentication and authorization data. Tomcat provides implementations for five different realms:

- DataSourceRealm (JNDI JDBC data source)
- JNDIRealm (LDAP accessed using JNDI)
- UserDatabaseRealm (In memory database)
- JAASRealm (Java Authentication and Authorization Service)

Each implementation can be activated by coding the appropriate *Realm* element in `server.xml`. The UserDatabaseRealm realm is already in place and is utilized by the `UserDatabase` resource to provide authentication and authorization data. This data is stored in `tomcat-users.xml` and is utilized for the manager web application. Credentials for other web applications can be placed in `tomcat-users.xml`. Alternatively, the DataSourceRealm can be employed and the roles and usernames can be stored in database tables. We will consider this important option in the following demo.

Demo 8.1: tomcat-security/Demos/DataSourceRealmElement.xml


```
1.    <Realm className="org.apache.catalina.realm.DataSourceRealm"
2.       connectionName="root" connectionPassword="rootpw"
3.       dataSourceName="jdbc/tomcatRealm" localDataSource="true"
4.       connectionURL="jdbc:mysql://localhost:3306/tomcatRealm" driver  ↵↵
               Name="com.mysql.jdbc.Driver" roleNameCol="rolename"
5.       userCredCol="password" userNameCol="username" userRoleTable="roles"
               userTable="users"/>
```

Code Explanation

The file above contains a `Realm` element that defines a DataSourceRealm for an authentication database. In an earlier lesson the attributes of this element were presented. Note that the driver class and database URL are specified in addition to the names of the user and roles tables and the column names of user name, password and role. Also note the `dataSourceName` attribute. The value of this attribute must match a resource JNDI name that is defined in a `Resource` element. Refer to `Demos/ResourceElement.xml` for an example.

This element can be placed with the scope of one of the following elements:

- <Engine> the realm is available to all web applications

- <Host> the realm is available to all web applications within this virtual host. It is not available to other virtual hosts.
- <Context> the realm is available only to the web application defined by this context descriptor.

To install this realm follow these steps:

- Shutdown the Tomcat server.
- Open `server.xml` in a text editor. Copy the Realm element from the Demos folder to the `Host` element. Change the realm database name from "tomcatRealm" to "tomcatRealmExercise". (Note: "tomcatRealm" has been implemented as the solution). Save the file.
- Start the Tomcat server. Watch the console messages to insure no exceptions are thrown due to an incorrect change to `server.xml`.

Demo 8.2: tomcat-security/Demos/web.xml

```
1.   <security-constraint>
2.    <web-resource-collection>
3.     <web-resource-name>Cool Garden Tools</web-resource-name>
4.     <url-pattern>/*</url-pattern>
5.     <http-method>GET</http-method>
6.    </web-resource-collection>
7.    <auth-constraint>
8.     <role-name>admin</role-name>
9.    </auth-constraint>
10.   </security-constraint>
11.   <login-config>
12.   <auth-method>BASIC</auth-method>
13.   <realm-name>Tomcat Security Team</realm-name>
14.   </login-config>
15.   <security-role>
16.   <role-name>admin</role-name>
17.   </security-role>
```


Code Explanation

The descriptor file shown above defines a **security constraint** for the Cool Garden Tools web application. The constraint serves to enforce authentication and authorization rules for accessing the website.

The elements are described below:

- **`security-constraint`** restricts access to the URL contained in the constraint to users belonging to the role indicated in the `auth-constraint` element.
- **`web-resource-collection`** encapsulates the URLs that have restricted access.
- **`web-resource-name`** provides the display name for the collection.
- **`url-pattern`** specifies a URL pattern that has restricted access. The path is relative to the context path of the web application. An asterisk (`*`) after the slash indicates any path will match the pattern. For each URL pattern, code a separate `url-pattern` element.
- **`http-method`** indicates the HTTP method associated with the incoming URL. Values are GET, POST, PUT or DELETE. For multiple method specification, code one `http-method` element for each method.
- **`auth-constraint`** encapsulates the authorization(s) required to access the resource(s).
- **`role-name`** specifies the role that has the authorization to access the resource(s). A user must be in this role to use the web page(s). More than one `role-name` may appear within the `auth-constraint` element.
- **`login-config`** encapsulates the method of prompting for the user's credentials.
- **`auth-method`** specifies the method of prompting for authorization data. BASIC indicates a built-in modal dialog box should be presented to the user (this requires no additional coding on the part of the developer or administrator. FORM indicates a custom form (JSP) will be presented to the user. The JSP must be specified in the `form-login-config` element.
- **`realm-name`** defines the name that is associated with this security realm. This name is displayed when the user is prompted for credentials using BASIC authorization.
- **`security-role`** indicates the presence of a security role to the web application.
- **`role-name`** specifies the name of the security role. This defines the role referenced in the security constraint. Only one `role-name` element may be present with a `security-role` element.

8.2. Java SecurityManager

❖ 8.2.1. Overview

The Java SecurityManager permits you to establish a customized security policy for an application. A security policy is defined in a **policy file** and dictates the permissions that control a program's access

to the file system, network sockets, etc. For Tomcat, the policy file is `CATALINA_BASE/conf/catalina.policy`.

❖ 8.2.2. Standard Permissions

The `CATALINA_BASE/conf/catalina.policy` file defines standard permissions; the following are examples:

- **java.lang.RuntimePermission**: Controls access to methods such as `System.exit()`
- **java.io.FilePermission**: Controls read/write permissions to files and directories
- **java.net.SocketPermission**: Controls access to network sockets

❖ 8.2.3. Tomcat Permissions

The `CATALINA_BASE/conf/catalina.policy` file also defines custom permissions:

- **`org.apache.naming.JndiPermission`** : Controls read access to JNDI file-based resources
- **`java.io.FilePermission`** Created dynamically by Tomcat to permit a web application to read its context root directory and read/write the working directory where generated servlets are stored

❖ 8.2.4. Starting Tomcat with a Security Manager Using the Default Policy File

To start Tomcat with a SecurityManager in place use the `-security` option when starting Tomcat: `catalina.sh start -security`.

8.3. Secure Socket Layer (SSL)

Secure Socket Layer (SSL) is a technology that establishes a secure client to server communication. This is achieved by the encryption of data from browser to server and from server to browser. Authentication is also a feature of SSL. Authentication is accomplished using **certificates**, a form of credentials. For example, the web server will send the browser a certificate to verify its identity. The data associated with the certificate includes the web server's business unit, company name and locale.

Note that SSL was pioneered by Netscape and later inspired TLS (Transport Layer Security) when SSL 3.0 was released.

The certificate is digitally signed and serves to verify the authenticity of the owner's identity. The following steps demonstrate how the client, in this case the web browser, initiates a secure connection to the server (e.g. Tomcat).

- The browser (client) requests SSL communication using `HTTPS` protocol.
- The server sends a response containing the server's digital certificate.
- The digital signature is decrypted by the browser using the server's **public key**.
- The browser encrypts a message using the server's public key and sends the request to the server.
- The server decrypts the message using the server's **private key**.

❖ 8.3.1. Certificates, TLS and HTTP/2

Certificates are usually obtained from **Certification Authorities** such as VeriSign. We will use the open source toolkit **OpenSSL** to create a certificate for testing TLS protocol communication with Tomcat server over HTTP/2. Tomcat 9 inaugurates support for HTTP/2 that currently is available in major browsers over TLS. HTTP/2 offers performance advantages including the ability to send resources to the browser when the HTML page is initially requested.

❖ 8.3.2. OpenSSL

OpenSSL is an open source toolkit for generating self-signed certificates for use over Transport Layer Service (TLS) and SSL protocols. Most major browsers will warn you about linking to a server with an OpenSSL certificate if the certificate is not digitally signed by industry-trusted certification authorities such as Verisign.

OpenSSL is available with the Apache Web Server or by downloading from open source sites such as SourceForge (https://sourceforge.net/projects/openssl).

To use OpenSSL on Tomcat 9 you must install the Apache Portable Runtime (APR) library at `https://apr.apache.org/download.cgi`. For more information on APR visit `https://apr.apache.org/`.

❖ 8.3.3. JSSE

JSSE (Java Secure Socket Extension) is an alternative to using OpenSSL. Certificates for use with JSSE can be build using the `keytool` utility that ships with the JDK. For more information on using JSSE

in Tomcat visit https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html#General_Tips_on_Running_SSL. (https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html#General_Tips_on_Running_SSL)

8.4. tomcat-users.xml

This XML file is located in the `conf` directory and serves as the "database" for the `UserDatabase` resource defined in `server.mxl`. This database is referenced by the `UserDatabase` realm that is also defined in `server.xml`. The manager application (discussed in a previous lesson) utilizes this realm for authentication. Because the manager has significant capability, the web application is shipped with web descriptor elements in place to restrict access to its web pages.

Sample roles and users are provided in `tomcat-users.xml` within comment tags. For example:

```
<role rolename="tomcat"/>

<user username="tomcat" password="tomcat" roles="tomcat"/>
```

Exercise 8: Creating DataSource Realm Authentication Database and Restricting Access to Cool Garden Tools Web Application

60 to 90 minutes

In this exercise you will create the `users` and `roles` tables necessary to implement the authentication and authorization model for the DataSource Realm that was activated in an earlier demo. Next, you will modify the `web.xml` in the Cool Garden Tools web application to restrict access to the website.

1. The `Exercises` directory contains the script to create the authentication and authorization tables. Open `RolesAndUsers.sql` in your text editor to view the SQL statements. The script currently populates the tables with one user and one role. Note that the column name for user name has to be identical in both tables. The SQL data type of each column must be character and must be set to a length sufficient to store user names and passwords. You can insert as many users and roles as necessary to implement your authentication model.
2. One user ("Nancy") has already been defined. This user is associated with the "admin" role. In your edit session, add the following lines after the insert statements already present in the file:

```
insert into users values ("Bill","catalina");
insert into roles values ("customer","Bill");
```

   The first insert statement creates a user "Bill" with password "catalina". The second insert statement places "Bill" in the role of "customer".
3. Save your changes.
4. Start the MySQL command line client (refer to the previous chapter for an example of starting the MySQL client). Create the database by entering the following command: `create database tomcatRealmExercise;`.
5. Enter the following command: `use tomcatRealmExercise`.
6. Run the SQL script by entering: `Exercises/RolesAndUsers.sql`.
7. Ensure that the statements ran correctly.

8. Shutdown the Tomcat server.
9. Open `web.xml` in the `CATALINA_BASE/webapps/Cool Garden Tools/WEB-INF/` in a text editor.
10. Using the demo file as a guide, add the necessary elements to restrict access to `/ToolDetail` and `/ShoppingCartAdd`. (**Hint:** code two `url-pattern` elements in the scope of `Cool Garden Tools Customer Pages` replacing the pattern `/*`).
11. Add the `role-name` of "customer" in the `auth-constraint`.
12. Add an additional `security-role` element containing a `role-name` of "customer".
13. Copy the `security-constraint` element (start tag through the end tag) so that now you have two identical `security-contraint` elements.
14. In the second `security-constraint` element, change the web resource name to "Cool Garden Tools Admin Pages".
15. Change the first `url-pattern` element to specify `/Vendors`. Delete the second `url-pattern` element.
16. Remove the `role-name` of "customer" from the `auth-constraint`. There should only be one role remaining, the role of "admin".
17. Save your changes.
18. Start the Tomcat server.
19. In your browser, go to `http://localhost:8080/CoolGardenTools`. Select a tool from the list that is displayed on the web page.
20. You should be prompted for user name and password. Enter "Bill" for the user name and "catalina" for the password. Click the **OK** button.
21. The detail page for the tool should now be displayed.
22. Add the item to your shopping cart. View the shopping cart to ensure the item has been added.
23. Now, change the location to `http://localhost:8080/CoolGardenTools/Vendors`. You should now receive an HTTP status of 403, "Access denied".
24. Empty browser cache so that active logins will be cleared from the browser's memory.
25. Go to `http://localhost:8080/CoolGardenTools/Vendors`. You should be prompted for user name and password. Enter "Nancy" for the user name and "tomcat" for the password.
26. The Vendors web page should now be displayed.

Solution

The `security-constraint` elements in `web.xml` for securing the Cool Garden Tools web application are shown below:

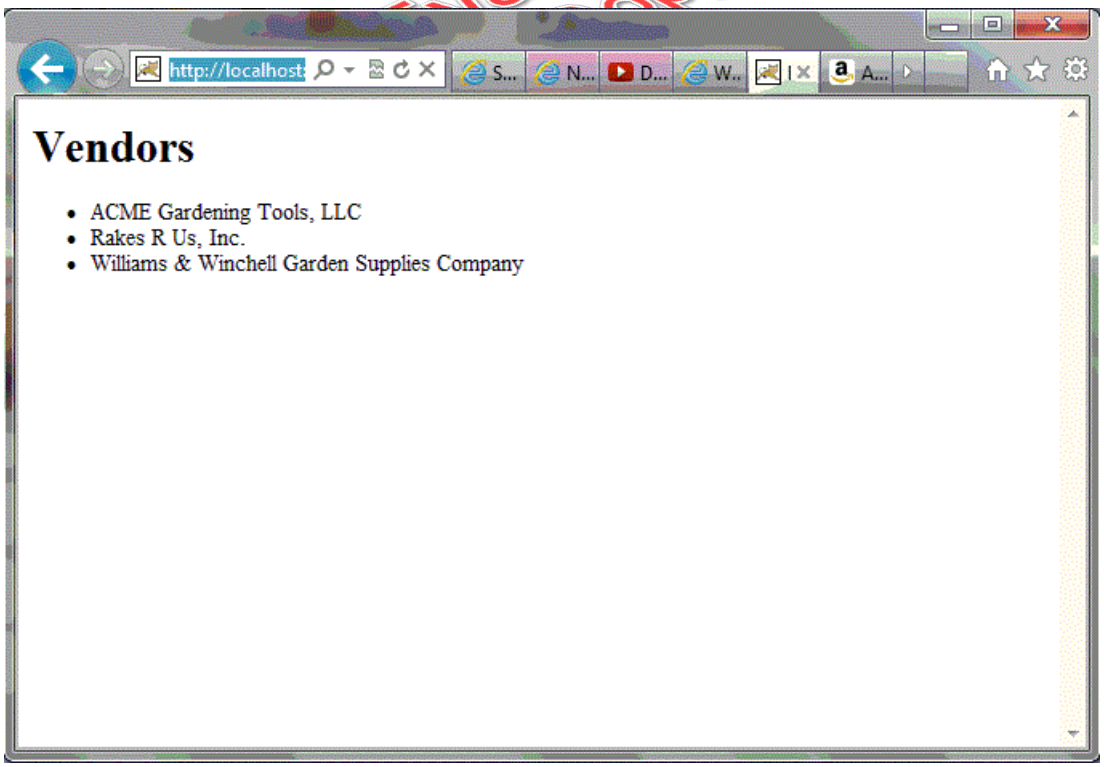
```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Cool Garden Tools Customer Pages</web-resource-name>
    <url-pattern>/ToolDetail</url-pattern>
    <url-pattern>/ShoppingCartAdd</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>customer</role-name>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Cool Garden Tools Admin Pages</web-resource-name>
    <url-pattern>/Vendors</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

The `login-config` and `security-role` elements in `web.xml` for securing the Cool Garden Tools web application are shown below:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Tomcat Security Team</realm-name>
</login-config>
<security-role>
  <role-name>customer</role-name>
</security-role>
<security-role>
  <role-name>admin</role-name>
</security-role>
```

Shopping cart displayed when authenticated as a customer:

Vendor list displayed when authenticated as an administrator:



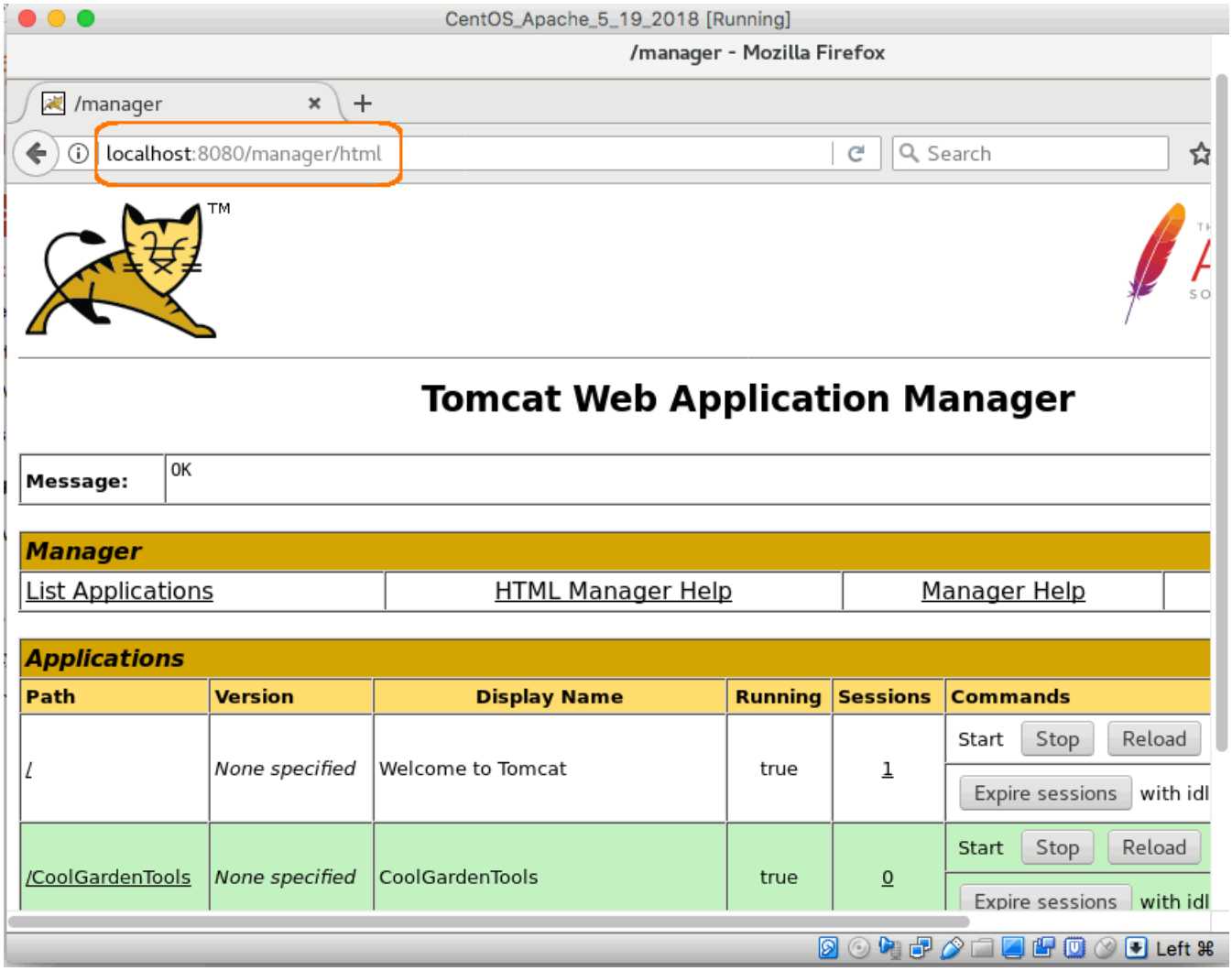
Exercise 9: Using DataSourceRealm for manager Application Authentication

30 to 45 minutes

In this exercise, you will configure Tomcat so that DataSourceRealm authentication is installed for the manager application.

1. During an earlier demo, you installed a DataSourceRealm to authenticate users for the Cool Garden Tools web application. In the previous exercise you created the authentication/authorization tables in MySQL. Because the realm element was coded in the scope of the `Host` element, this realm will override the in-memory database utilized for the manager application and therefore deactivate the user names and roles in `tomcat-users.xml`.
2. Authentication (authorization enforced by the roles table) must be achieved using the DataSourceRealm database if we are to retain the security model that is protecting the Cool Garden Tools web application.
3. Shutdown the Tomcat server. In your text editor open the `RolesAndUsers.sql` script provided in the `Exercises` directory. (Alternatively, you can modify `RolesAndUsers.sql` from the previous exercise).
4. Insert into the roles table a row associating "Nancy" with the role of "manager-gui".
5. Save your changes.
6. Start the MySQL database server.
7. Using the MySQL command line client, run the `RolesAndUsers.sql` script. Ensure that the SQL statements execute successfully.
8. Start the Tomcat server.
9. In your browser, go to `http://localhost:8080/manager/html`.
10. When prompted for credentials, enter the user name "Nancy" and the password "tomcat". Click **OK**. The manager page should now be displayed.
11. Ensure that you can utilize manager functions by stopping, and then starting, the Cool Garden Tools web application.

Solution



Conclusion

In this lesson, you have learned:

- The security considerations for web applications.
- The basics of the Java SecurityManager.
- Secure Socket Layer.
- Using DataSourceRealm for Web application security.
- The purpose of tomcat-users.xml.

LESSON 9
Logging

Topics Covered

☑ The purpose of logging.

☑ Logging techniques in Tomcat and web applications.

☑ Tomcat's implementation of Apache Commons Logging.

Introduction

Logging is the process of directing informational, debugging, warning and severe messages to a destination file. Tomcat uses an implementation of the Apache Commons Logging library, a thin wrapper on top of a logging framework. The logging framework Tomcat implements is `java.util.logging`. Web applications can use their own copies of the Apache Commons Logging library.

Tomcat also provides Valve implementations for **request dumping** and **HTTP access**. Request dumping routes detailed information on incoming HTTP requests to a log file. HTTP access logging can be tailored to capture data such as HTTP method, IP, session attributes and other items.

Apache's `log4j` can be installed for use by Tomcat. The steps for installing `log4j` will be presented in this lesson.

9.1. Logging Overview

Logging is potentially beneficial to administrators and developers to verify the operation of server components and web applications. In addition, logging is a tool to assist administrators and developers in debugging situations.

Tomcat by default provides logging via its own implementation of commons logging called **JULI** (Java Utility Logging Implementation) and is hard-coded in `org.apache.juli.logging.LogManager`.

Tomcat utilizes **handlers** to write data to log files. Multiple handlers of a single class (e.g. `FileHandler`) can be instantiated by assigning a prefix to the handler name (e.g.

1catalina.org.apache.juli.FileHandler). The Catalina handler logs messages from the Catalina engine whereas the localhost handler logs data from the virtual host set up within Catalina. Two additional handlers are provided for the manager and host-manager web applications. Various properties of each handler can be changed in the CATALINA_BASE/conf/logging.properies file:

- **org.apache.logging.juli.FileHandler.level** specifies the logging level; options are FINEST, FINER, FINE, DEBUG, INFO, WARNING, ERROR and SEVERE. For example, setting the level to WARNING will cause the handler to write warning messages and higher levels (e.g. ERROR) to the log file. To turn off all logging, specify a level of OFF; a level of ALL indicates that all messages are to be written to the log.
- **org.apache.logging.juli.FileHandler.directory** sets the directory where the log file will be written.
- **org.apache.logging.juli.FileHandler.prefix** indicates the prefix that is appended to the log file name.

The logging.properties file assigns the Catalina handler and the console handler as **root logger** handlers using the .handlers property. The root logger generates messages pertaining to Coyote startup, Catalina initialization and web deployment activity. Here is the assignment in logging.properties:

```
.handlers          =          1catalina.org.apache.juli.FileHandler,
java.util.logging.ConsoleHandler
```


Therefore messages produced by the root logger are routed to the Catalina log file and to the console. To prevent this redundancy, you can remove one of the handlers from the assignment. For example:

```
.handlers = 1catalina.org.apache.juli.FileHandler
```

Now, root logger messages will appear only on the Catalina log. Messages can still be written to the console, e.g. Java programs that write to System.out.

The properties file is a clear text file. To configure the HTTP access log or the request dumper Valve, you will need to modify server.xml. HTTP access log configuration is discussed in the following demo.

Demo 9.1: tomcat-logging/Demos/accessLogLinux.xml

```
1.   <Valve className="org.apache.catalina.valves.AccessLogValve"
2.     directory="/logsDemo" test="erase this attribute, just a test"
3.      prefix="localhost_access_log." suffix=".txt"
4.     pattern="%H %m %s %v %{shoppingCart}s"
5.     resolveHosts="false"/>
```

Code Explanation

An **access log** is composed of HTTP messages that are sent from clients to the server. Tomcat is shipped with an implementation of `org.apache.catalina.valves.Valve` that writes the log to a destination file. The demo file above contains the `Valve` element that is present in `server.xml`. The attributes in the demo are described below:

- **className** name of the implementing Java class.
- **directory** directory where log file will be stored. This value has been modified to a directory external to CATALINA_HOME (original value was `logs`.
- **prefix** log file name prefix.
- **suffix** extension of the file name.
- **className** name of the implementing Java class.
- **pattern** log record layout; `common` and `combined` are preset sequences of pattern codes. This can be replaced with your own string of pattern codes. For example, `"%H %m %s %v %{shoppingCart}s"` will display the host protocol, HTTP method, HTTP status, server name, and the HTTP session variable `shoppingCart`.
- **resolveHosts** boolean value that specifies if a DNS lookup should be performed to convert the remote IP to a hostname. A value of `true` indicates the lookup should be performed; `false` indicates that the lookup is not to be performed. The default value is `true`.

Documentation on the Access Log valve can be found at `http://tomcat.apache.org/tomcat-9.0-doc/config/valve.html#Access_Log_Valve (https://tomcat.apache.org/tomcat-9.0-doc/config/valve.html#Access_Log_Valve)`.

9.2. Web Application Logging Techniques

Web applications can perform their own logging. Three common frameworks are `java.util.logging`, `javax.servlet.ServletContext` and `org.apache.log4j`.

❖ 9.2.1. java.util.logging

This API is provided with Java Standard Edition. Applications use the `Logger` class methods to log messages. For example, the `severe` method causes the logging of a message at the level of severe (**Note:** internally this level is mapped to an integral value of 1000, the highest value of any log level).

❖ 9.2.2. javax.servlet.ServletContext

Web applications can log using the `ServletContext.log(String)` method. This API is generally disfavored as it's not as robust as other logging APIs.

❖ 9.2.3. log4j

Apache's `log4j` is a full-function logging API that has gained widespread acceptance in the Java community. Web applications can utilize `log4j` by storing the `log4j.jar` under *`WebContextPath`*`/WEB-INF/lib`. You can also configure Tomcat to utilize `log4j`. Follow these steps:

1. Download the log4j zip file from `https://logging.apache.org/log4j/1.2/download.html`. Extract the contents into a directory. Copy `log4j.jar` into `CATALINA_HOME/lib` (**Note:** the actual jar file name will be appended with version and release numbers).
2. Create a file named `log4j.properties` and save it under `TOMCAT_HOME/lib`. To look at a sample, go to `http://tomcat.apache.org/tomcat-7.0-doc/logging.html` (`https://tomcat.apache.org/tomcat-7.0-doc/logging.html`). The `log4j.properties` file defines **appenders** and associates Tomcat loggers (e.g. Catalina) with an appender. For example, the **daily rolling file appender** writes to a log file and rolls over the log at a frequency determined by the property `DatePattern`. The frequency can be set to rollover every hour, at midnight every day, every week or every month. Prior to rollover, a backup of the log is created. The **console appender** supports properties that determine how data is stored in the console log. Logging levels consistent with `java.util.logging` are supported.
3. Download `tomcat-juli.jar` and `tomcat-juli-adapter.jar` from the Tomcat website (these are stored in the "extras" directory).
4. Place `tomcat-juli.jar` in `CATALINA_HOME/bin` (this will overwrite the default jar file already present in this folder).
5. Place `tomcat-juli-adapter.jar` in `CATALINA_HOME/lib` (this will overwrite the default jar file already present in this folder).
6. Remove the `logging.properties` file from the `CATALINA_HOME/conf` directory to prevent the generation of zero-length log files from the default configuration.
7. Start the Tomcat server.
8. Verify that logs are written according to the specifications in the properties file.

Exercise 10: Logging with java.util.logging

20 to 30 minutes

In this exercise, you will modify the logging properties file to filter the output to the Catalina log. You will also rename the log file.

1. Shut down the Tomcat server. Create a directory under the File System root named `logExercise`. Change the permissions so that all users can create files in the directory.
2. Edit the `CATALINA_BASE/conf/logging.properties` file and locate the Catalina entries. Change the `FileHandler.level` to **`INFO`**. Change the `FileHandler.directory` to `/logExercise`. Change the `FileHandler.prefix` to **`catalinaExercise`**.
3. Start the Tomcat server. Verify the log file has been created in the `logExercise` directory and has the assigned prefix.
4. View the contents of the log. Note that only `INFO` messages appear.
5. Shut down the server and delete the log file. Edit the `CATALINA_BASE/conf/logging.properties` file and change the level to `WARNING`. Save the file. Start the server. The log file should be empty because `INFO` messages are at a lower log level than `WARNING` messages.

Exercise 11: Formatting the Access Log

30 to 45 minutes

In this exercise, you will activate the access log and try different formats for the log.

1. Shut down the Tomcat server. Create a directory named `accessLogExercise` under the File System root. Change the permissions so that all users can create files in the directory.
2. Delete all log files in the `CATALINA_BASE/conf/logs` folder.
3. Open `CATALINA_BASE/conf/server.xml` in your text editor. Locate the access log `Valve` element in the scope of the `Host` element. Specify a destination directory of `/accessLogExercise`. Specify a prefix of `localhost_access_log` with a suffix of `.log`.
4. Set the `pattern` attribute value to `"%H %m %s %v %{shoppingCart}s"`. Save the changes. Keep the text editor open so that you can make additional modifications to the attribute value.
5. Start the Tomcat server. In your browser, go to `http://localhost:8080/CoolGardenTools`. Select a tool and add the tool to your shopping cart.
6. Navigate to the `accessLogExercise` directory. Check the name of the log file to ensure it is consistent with the Valve attribute values.
7. Open the log file in your text editor so you can view the contents of the file.
8. Observe how the data is presented in accordance with the pattern you specified: HTTP protocol, method and status code followed by the host name and session data. The session data is stored in an ArrayList, `shoppingCart`. The elements are enclosed in square brackets.
9. Return to the web application and add another item to your shopping cart. Review the log and note that two elements are stored in the ArrayList now.
10. Shutdown the server. In your text editor, change the `pattern` attribute to `"%H %m %s %v %u"`. The `%u` pattern code specifies that the user name is to be displayed. Save your changes and leave the text editor open.
11. Start the server and go to the Cool Garden Tools web application in your browser. Check the log. You should see the user name listed after the host name.
12. Stop the server and return to your text editor. Change the `pattern` attribute to `"%H %m %s %v %U"`. Note that you have modified the last pattern code from a lowercase 'u' to an uppercase 'U'. This pattern code specifies that the URL path is be displayed. Save your changes.
13. Start the server and go to the Cool Garden Tools web application in your browser. Check the log. You should see the URL listed after the host name.

Solution

Valve element with pattern to display HTTP protocol, method, HTTP status, host name and `shoppingCart` session attribute:

```
<Valve                    className="org.apache.catalina.valves.AccessLogValve"
directory="/accessLogExercise"   prefix="localhost_access_log."   suffix=".log"
pattern="%H %m %s %v %{shoppingCart}s" resolveHosts="false"/>
```

Log output using the settings presented above:

```
HTTP/1.1 GET 200 localhost
[ID: 7 Description: Shovel with Grip Handle Price: $34.49]
HTTP/1.1 GET 304 localhost
[ID: 7 Description: Shovel with Grip Handle Price: $34.49]
```

Notice the HTTP status **304**. This is not actually an error but rather indicates to the browser that the resource (web page) has not been modified since the last request made by the browser. Therefore the browser can use its cached copy of the resource.

Valve element with pattern to display HTTP protocol, method, HTTP status, host name and user name:


```
<Valve                    className="org.apache.catalina.valves.AccessLogValve"
directory="/accessLogExercise"   prefix="localhost_access_log."   suffix=".log"
pattern="%H %m %s %v %u" resolveHosts="false"/>
```

Valve element with pattern to display HTTP protocol, method, HTTP status, host name and the URL path:

```
<Valve                    className="org.apache.catalina.valves.AccessLogValve"
directory="/accessLogExercise"   prefix="localhost_access_log."   suffix=".log"
pattern="%H %m %s %v %U" resolveHosts="false"/>
```

Conclusion

In this lesson, you have learned:

- The purpose of logging.
- The techniques available to Tomcat and web applications for logging.
- Tomcat's implementation of Apache Commons Logging.

LESSON 10

Monitoring and Performance Tuning Tomcat

Topics Covered

☑ The process of monitoring.

☑ Tuning Tomcat for optimal performance.

☑ Using JMX MBeans in monitoring.

☑ Tools that assist in monitoring.

Introduction

Delivering the fastest possible responses to web clients is the fundamental goal of performance tuning. Monitoring is the process of observing critical resource activity such as heap memory usage and thread allocations. Performance tuning is the process of adjusting parameters such as minimum heap memory size and maximum thread allocations after monitoring these resources to achieve optimal performance.

The JVM utilizes heap memory to store objects instantiated by application programs. Non-heap memory is reserved for class (static) methods and fields, method and constructor procedural code, threads and other memory requirements of the JVM. For example, the JVM might elect to generate native machine code using the **Just-In-Time** compiler based on method and looping volume for a particular class. The storage for the operation codes is taken from the non-heap memory.

10.1. Tomcat

The focus of performance tuning in Tomcat is adjusting the number of threads in the Catalina service and the size of cache memory. Tomcat 10 supports **NIO** (Nonblocking I/O). NIO gives you additional attributes under the NIO implementation of Connector. You control these parameters by setting attribute values within `server.xml`. The following list is a representative sample of important attributes.

- `autoDeploy` attribute of `Host` element: when set to true Tomcat runs a background thread to periodically monitor `/webapps`; setting the value to false will prevent execution of this thread.

- `enableLookups` attribute of `Connector` element: if set to true this value causes a DNS (domain name service) lookup on calls to `request.getRemoteHost()` to return the host name of the remote client. The default value is true. Consider setting this to false to eliminate the overhead of a DNS lookup when `request.getRemoteHost()` is invoked. The method call will return a string representing the IP address if this attribute is set to false.
- `maxThreads` attribute of `Connector` element: this value determines the maximum number of simultaneous requests that can be handled. In coming requests will then be queued. The default value is 100. A higher value might adversely impact performance during peaks of request activity. A lower value might promote better performance for existing requests but may also cause more requests to be queued and the possibility of rejecting requests if `acceptCount` is exceeded. The default is generally acceptable.
- `tcpNoDelay` attribute of `Connector` element: when set to true (the default) Tomcat will set TCP_NODELAY on the server socket. This will in general improve performance because buffering of TCP packets is bypassed and therefore the packet is transmitted immediately.
- `processorCache` attribute of `Connector` element (NIO implementation) specifies how many Http11NioProcessor objects are to be cached in an effort to improve performance. Default is 200 (the same number as `maxThreads`.
- `socket.bufferPool` attribute of `Connector` element (NIO implementation) specifies how many NioChannel objects to cache in an effort to reduce garbage collection. Default is 500.

10.2. JVM

The following is a representative sample of JVM options that are important to performance tuning. In particular, note the options that apply to garbage collection. The garbage collector (GC) has been refined over successive releases of the JVM in order to reduce performance degradation germane to suspending the execution of application code while the GC is running. For example, **generational** garbage collection takes advantage of various memory pools that are reserved for "young" and "old" objects. This reduces the **marking** effort that must precede a **sweep**,the process of freeing the memory held by an object.

For detailed information, visit the Oracle white paper on tuning at `https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`

- `-Xincgc` requests that incremental garbage collection be performed. This causes the garbage collector to collect only a percentage of the heap and should lessen the overall time spent performing garbage collection.

- `-XX:+UseParallelGC` requests the parallel garbage collector. One thread is established for each CPU so that concurrent garbage collection can be achieved. This parameter is automatically set for multi-CPU computers that have at least 2GB of RAM. This parameter can be omitted if `UseParallelOldGC` is specified (described below).
- `-XX:+UseParallelOldGC` requests the parallel garbage collector for the old generation. The old generation is a pool in heap memory where objects are placed after surviving several garbage collection attempts. If this parameter is specified then `UseParallelGC` can be omitted because `UseParallelOldGC` implements all the functionality of `UseParallelGC`.
- `-XX:+UseConcMarkSweepGC` requests the concurrent mark sweep (CMS) garbage collector. This is the alternative GC policy to concurrent garbage collection and results in the elimination of long garbage collection pauses, at the expense of some overall performance.
- `-Xms` specifies the minimum heap size as a number. For example, `-Xms 100m` allocates 100 megabytes of heap memory, while `-Xms 2g` allocates 2 gigabytes of heap memory. The larger the minimum heap size, the greater the probability that additional memory will not be requested frequently. Consider setting minimum and maximum to the same value to eliminate the time the JVM requires to grow the heap memory from the minimum size up to the maximum size.
- `-Xmx` specifies the maximum heap size as a number. For example, `-Xmx 100m` allocates 100 megabytes of heap memory, while `-Xmx 2g` allocates 2 gigabytes of heap memory. This option indicates the maximum amount of memory that can be allocated. Consider setting minimum and maximum to the same value to eliminate the time the JVM requires to grow the heap memory from the minimum size up to the maximum size. The possible drawback is that when garbage collection does run, a significant lag time could result.
- `-XX:MaxPermSize=n` specifies the maximum size of the permanent generation area of non-heap memory. Objects stored in this area include class files loaded by the class loaders. For example, 100m allocates 100 megabytes of permanent generation memory, while 2g allocates 2 gigabytes of memory. The default is 64 megabytes on Java SE 6 for Solaris Sparc. This size should be increased for most production servers.

To change a JVM option, set the `JVM_OPTS` parameter in `setenv.sh`. For example, to set the maximize size of non-heap memory to 1 gigabyte, code the following:

```
export JVM_OPTS=-Xmx1g
```

10.3. JMX (Java Management Extensions)

JMX technology was introduced in Java 5. JMX provides a way of managing resources through **managed beans** (**MBeans**). A given resource can be measured by one or more MBeans. Monitoring can be accomplished by viewing properties of managed applications and/or services. Management is the process of changing a property value and then observing the results of the modification. MBeans provide a convenient interface for both monitoring and management. MBeans expose **attributes** that contain information about the resource and **operations** that can be called to change specific resource values.

10.4. JMX MBeans in Tomcat

Tomcat is shipped with a variety of MBeans that can be exposed through Apache `ant` or a tool such as `jconsole`. The MBeans can be categorized as follows.

❖ 10.4.1. Engine

Attributes included are `baseDir` and `defaultHost`. Operations include **start** and **stop**.

❖ 10.4.2. JKMain

This MBean exposes attributes and operations applicable to the Apache Web Server to Tomcat connector. We will look at the JK connector in a later lesson.

❖ 10.4.3. String Cache

This MBean exposes attributes such as `accessCount` and `cacheSize`.

❖ 10.4.4. Server

This MBean exposes attributes such as shutdown `port` and `serverInfo`.

❖ 10.4.5. Users

This MBean provides `username` and `password` data from `tomcat-users.xml`. In addition, attributes are provided to display role and user database information.

10.5. Configuring Tomcat to use MBeans

In a previous lesson you learned about the `setenv.sh` script. The file provided contained the Catalina properties necessary to activate Tomcat's MBeans. We will visit this file again and explain the properties.

Demo 10.1: tomcat-monitoring-and-performance-tuning/Demos/setenv.sh

```
1.    export CATALINA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxre  ↵↵
              mote.port=9009 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.man  ↵↵
              agement.jmxremote.authenticate=false"
2.    export JAVA_OPTS=-Xms10m
```

Code Explanation

The `-D` option on the `java` application launcher is utilized to set a **system property**. The properties set in the example above activate JMX within Catalina so that an external (remote) Java program (e.g. `jconsole`) can access MBeans defined in the application (Catalina). The properties are described below:

- **`com.sun.management.jmxremote`** enables JMX access and registers the JVM MBeans. Client applications can monitor a local JVM (a JVM running on the same machine) or a remote JVM.
- **`com.sun.management.jmxremote.port`** assigns a port to accept local and remote requests for MBean information. For a remote request (e.g. when using `jconsole`) specify hostname (e.g. `localhost`) and this port number.
- **`com.sun.management.jmxremote.ssl`** boolean value indicating if SSL (Secure Socket Layer) protocol is enabled on the port; a value of `false` indicates SSL is not enabled; a value of `true` indicates SSL is enabled. SSL is enabled by default when remote monitoring is activated.
- **`com.sun.management.jmxremote.authenticate`** boolean value indicating if password authentication is enabled; a value of `false` indicates authentication is to be disabled; a value

of `true` indicates authentication is to be activated. Password authentication is enabled by default when remote monitoring is activated.

- **`com.sun.management.jmxremote.password.file`** name of the file that contains the passwords for roles; the roles are defined in the file referenced by `com.sun.management.jmxremote.access.file`. A template for the password file can be found in `JAVA_HOME/jre/lib/management/jmxremote.password.template`. This property is ignored if `com.sun.management.jmxremote.authenticate` is set to false.
- **`com.sun.management.jmxremote.access.file`** name of the file that contains roles controlling access to the JMS port. A role can be assigned specifying access is "readonly" or "readwrite". A sample access file can be found in `JAVA_HOME/jre/lib/management/jmxremote.access`. This property is ignored if `com.sun.management.jmxremote.authenticate` is set to false.

For more information on securing the JMX port visit the Oracle documentation at `https://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html`

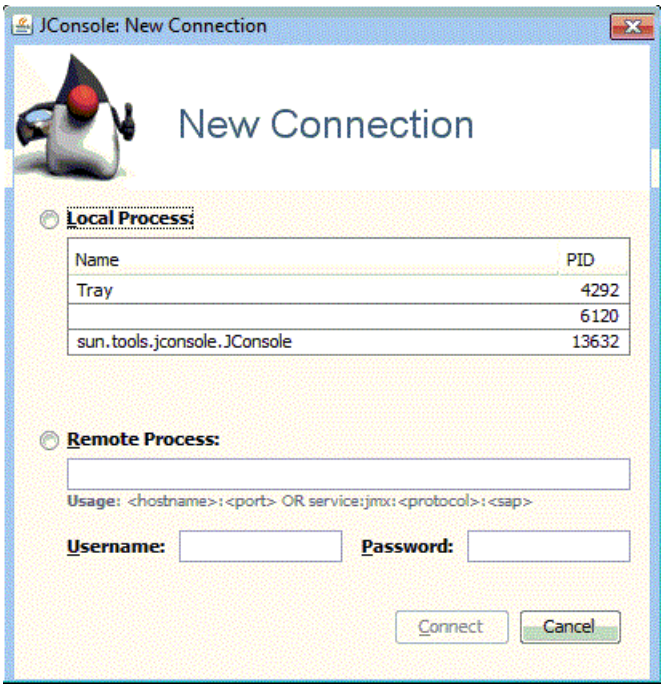
10.6. Accessing MBeans

Although `ant` targets can be utilized to access MBean attributes and operations, visual tools offer a more convenient interface. We will consider three such tools here.

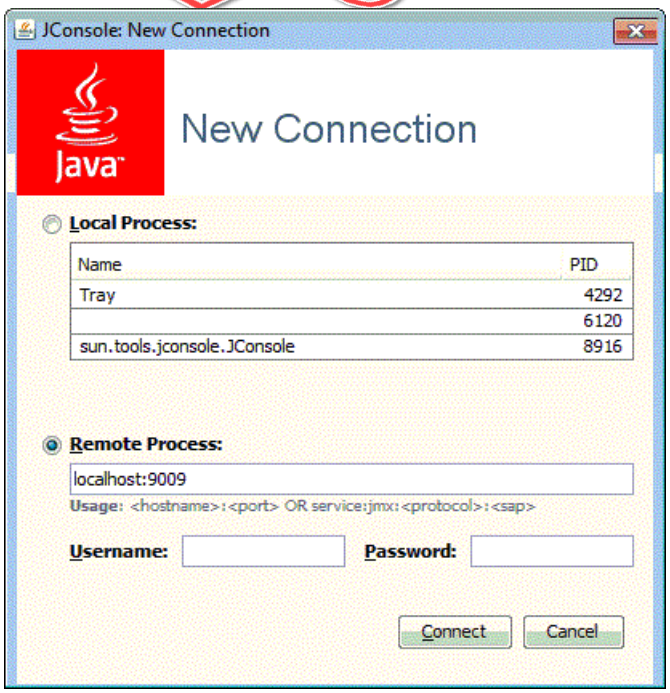
❖ 10.6.1. jconsole

The `jconsole` tool is shipped with JDK 1.5 and higher. This tool displays JVM memory and thread data as well as MBean attributes and operations. You will gain experience with this tool in the lesson exercise.

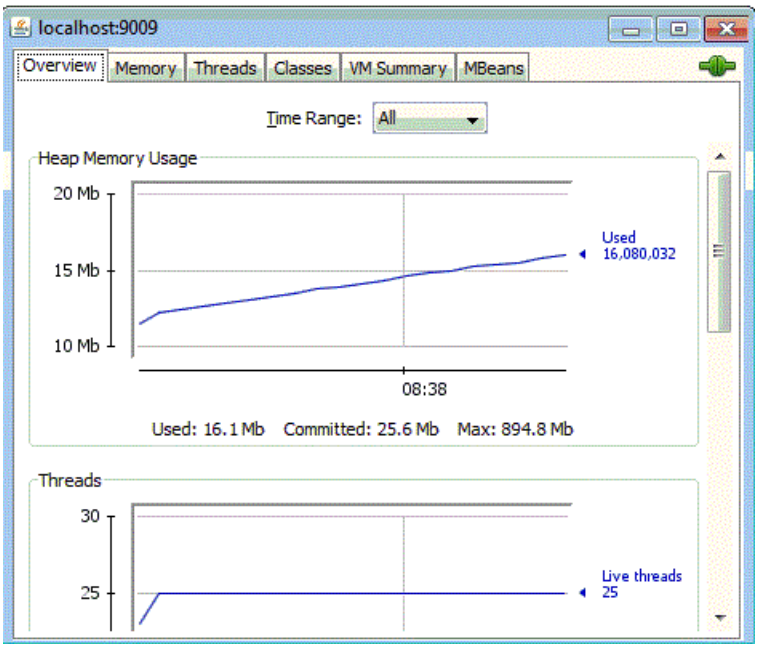
To run `jconsole` open a command prompt and type `jconsole`. Hit **Enter**. The connection page should be displayed:



Any local JVMs that are detectable by `jconsole` will be displayed. If the JVM you wish to connect to is not displayed, then you can attempt a remote connection by entering the host name and the port number (specified in `setenv.bat`) as follows:



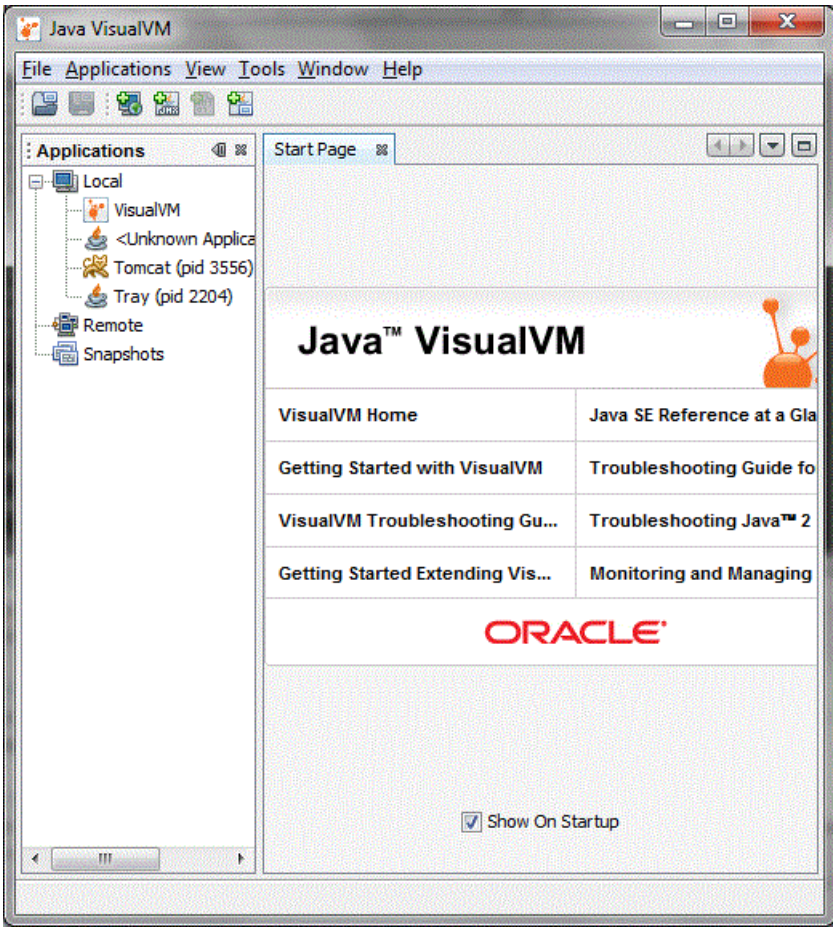
If the connection is successful, `jconsole` will display a tabbed pane presentation of the corresponding JVM:



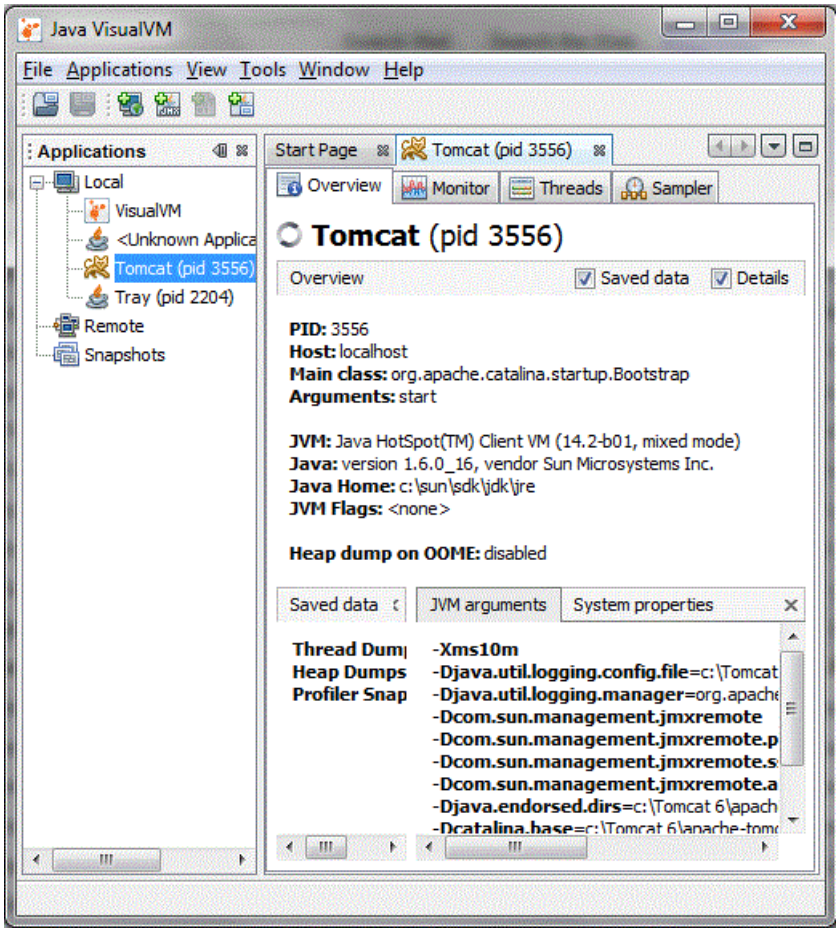
❖ 10.6.2. jVisualVM

The **Java Visual VM** tool was included with JDK 1.6 and 1.7. For later releases of Java, you can download at `https://visualvm.github.io/download.html`. The user interface includes tabs for JVM information, monitors, threads and a CPU and memory sampler. A navigation pane is provided as well.

Start the Tomcat server and open a command prompt window. Type `visualvm` on the command line and press **Enter**. The start page should appear.

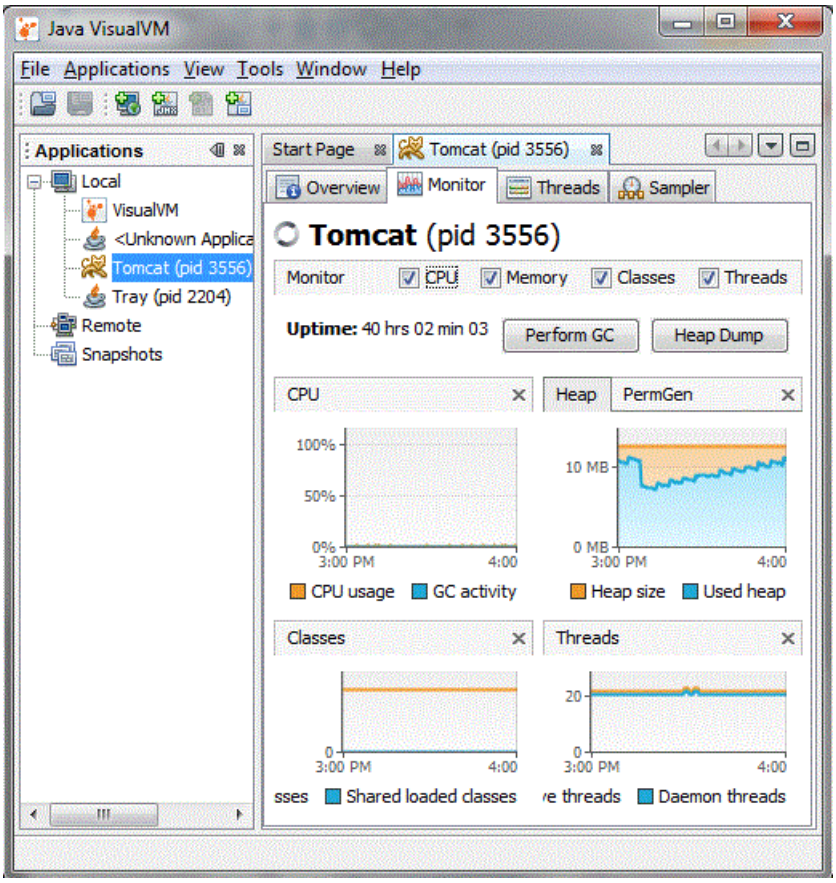


In the **Applications** pane, click `Tomcat`. The overview tab for the Tomcat instance will be displayed:

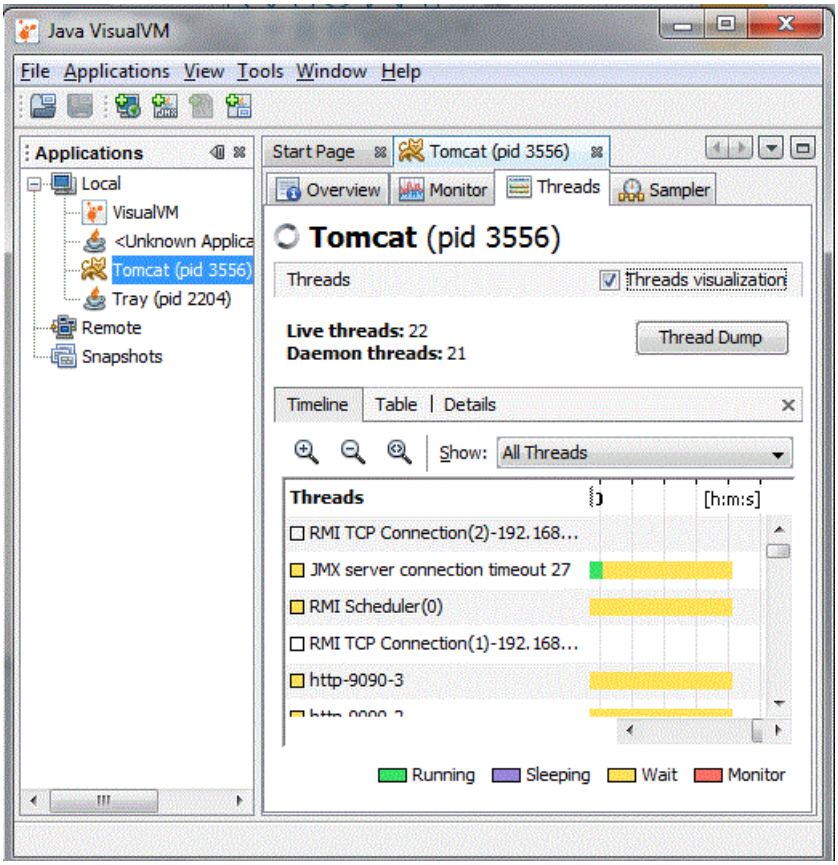


Note that the JVM arguments are displayed. These arguments were established by the `setenv.bat` file illustrated earlier. Click the `System properties` tab. Now you can see the system properties that are built in to the JVM.

Click the `Monitor` tab. The monitor view will be displayed. Notice that CPU, memory, classes and threads graphs are shown.



Click the `Threads` tab. The threads view will be displayed. Notice that CPU, memory, classes and threads graphs are shown.



All of the threads operating on the Tomcat JVM are displayed and colorized to indicate the thread state.

❖ 10.6.3. PSI Probe

This monitoring tool is designed to replace the Tomcat manager application and was developed from the Lambda Probe. To download PSI Probe, go to `https://github.com/psi-probe/psi-probe/releases`

Exercise 12: Using jconsole to Monitor and Manage Tomcat

25 to 40 minutes

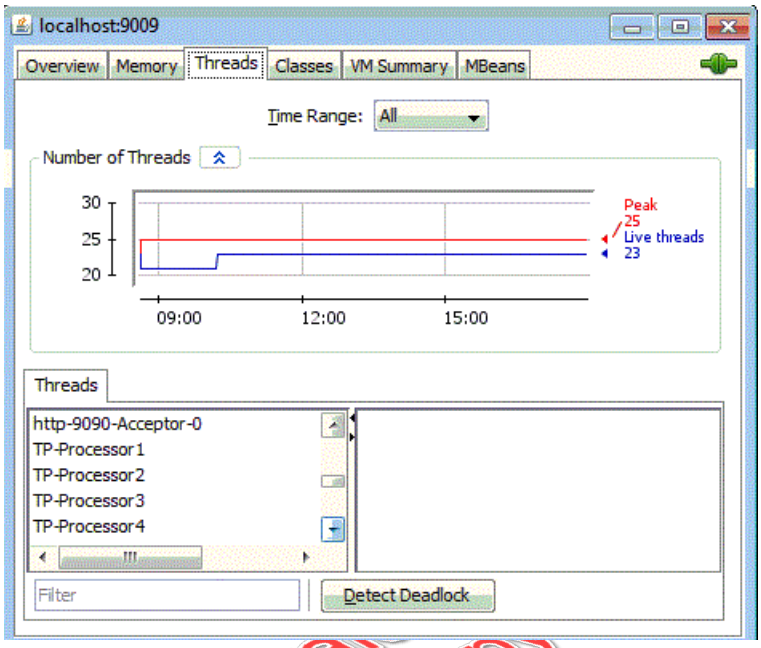
In this exercise, you will monitor Tomcat using jconsole and apply changes to the Tomcat setup.

1. Shut down Tomcat. Configure Tomcat to use JMX MBeans using the modifications to `setenv.bat` shown in the demo file presented earlier in this lesson.
2. Start Tomcat.
3. In a terminal window start `jconsole`. From the New Connection panel, select `org.apache.catalina.startup.BootStrap start`. Click the `Connect` button.
4. Select the Threads tab. Observe the peak number of threads. How many live threads do you see? Select some of the threads displayed at the bottom of the panel. Try to determine the purpose of the threads you select.
5. Click the Memory tab. By default, heap memory usage is displayed. You can display the non-heap memory by selecting the appropriate entry in the dropdown list box at the top of the panel. Note the lowest memory amount listed on the graph for heap memory. Under the graph you should see a "Committed" entry. What value is displayed?
6. In a browser, go to the "Cool Garden Tools" web application. Spend a minute or so selecting some of the tools and add those tools to your shopping cart.
7. Return to the jconsole session.
8. Click the Memory tab. Note the lowest memory amount listed on the graph for heap memory, as you did earlier. Under the graph you should see a "Committed" entry. Has this value changed since you last time you monitored this page?
9. Click the MBeans tab. In the tree view along the left hand side of the window, Select "Catalina | Connector | 8080 | Operations". Under Operations select "stop". You will see the "stop" operation expanded in the right hand panel.
10. Click the "stop" button at the top of the panel. This will invoke the stop operation on the HTTP connector. A pop up box should appear, confirming the operation is successful.
11. In your browser, try to invoke any web application. What happens?
12. Return to jconsole. Select the "start" operation in the left hand panel. Click the "start" button. This will restart the HTTP connector.
13. Return to the browser. Did the web application run?

14. If you need to temporarily stop traffic through your server to alleviate low memory errors, this technique can be beneficial.
15. Click the MBeans tab. In the tree view along the left hand side of the window, Select "Catalina | Deployer | localhost | Operations". Under Operations select "isDeployed". You will see the "isDeployed" operation expanded in the right hand panel.
16. The "isDeployed" operation accepts one parameter, the web application context. The operation returns true if the web application has been deployed.
17. Over type "String" within the parentheses with "/FormDemo". Click the "isDeployed" button. A pop up box should appear with the value of "true" indicating that the application has been deployed.
18. We'll presuppose the "FormDemo" web application is suspected of causing performance issues. Accordingly, we wish to temporarily stop the web application.
19. Click the MBeans tab. In the tree view along the left hand side of the window, Select "Catalina | WebModule | //localhost/FormDemo | none | none |Operations". Under Operations select "stop". You will see the "stop" operation expanded in the right hand panel.
20. Click the "stop" button at the top of the panel. This will invoke the stop operation on the "FormDemo", resulting in a pop up box informing us the method executed successfully.
21. In your browser, try to invoke the web application. What happens?
22. In a real world situation, we would monitor Tomcat to determine if performance (i.e., overall response times) have improved. If so, then we would work with web developers to find out what issue(s) the "FormDemo" web application is experiencing and apply program fixes.
23. To start the "FormDemo" web application, select the "start" operation under the Operations branch.
24. Invoke the "FormDemo" web application. You should see the form displayed on the page.

Solution

The threads display is shown below:



The following is a sample list of the types of threads that will be observed:

- **TP Processor** for communication with database connection pooling and other external agents.
- **RMI** for Remote Method Invocation interaction with Java classes.
- **HTTP-*port*** for HTTP communication using Coyote.
- **GC daemon** representing the execution of the garbage collector.

Conclusion

In this lesson, you have learned

- About the process of monitoring
- How to tune Tomcat for optimal performance
- How JMX MBeans can be useful in monitoring
- Some of the tools that assist in monitoring

LESSON 11
Clustering

Topics Covered

☑ Clustering for replication and load balancing.

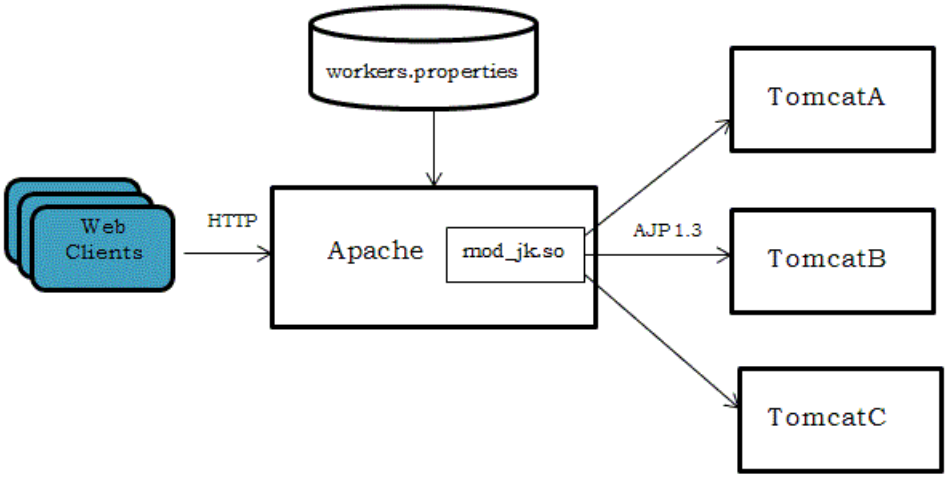
☑ Clustering and front end proxy.

☑ Load balancing.

Introduction

Clustering is the technique of running more than one Tomcat instance in a coordinated manner. The purpose of clustering is twofold: first, to implement **load balancing** and second, to achieve **session replication**. Load balancing implies distributing incoming requests across more than one Tomcat instance. Session replication is the ability to duplicate session attributes in a different Tomcat instance so if one instance fails the session can be resumed transparently from the user's point of view.

Tomcat 9.0 does not provide a load balancer. The Apache web server can be utilized as a front end proxy to implement load balancing across two or more Tomcat instances.



Web clients (browsers) send HTTP requests to Apache. Apache routes requests for Tomcat web applications to a particular Tomcat server. The servers are known as **workers** and are defined to Apache in its configuration file.

11.1. Using Clustering for Replication and Load Balancing

To configure Tomcat for clustering, uncomment the sample `<Cluster>` element in `CATALINA_BASE/conf/server.xml`. The only implementation provided by Tomcat is `SimpleTcpCluster` but the default configuration is a good starting point and that is what we will consider in this lesson.

When you activate `SimpleTcpCluster` the cluster architecture is automatically configured. The following elements are installed:

- **`Manager`** specifies `org.apache.catalina.ha.session.DeltaManager` as the manager to handle session replication.
- **`Channel`** establishes abstract endpoint, or socket, that a member of the cluster can use to send and receive session replication data. Channels are managed by the **Apache Tribes communication framework**.
- **`Membership`** specifies the address and port utilized for multicast communication among the members of the cluster.
- **`Receiver`** indicates the TCP receiver of replicated session data from cluster members.
- **`Sender`** specifies the TCP component that sends replicated session data to cluster members.
- **`Interceptor`** enables a component that can alter characteristics of a channel.
- **`Valve`** defines a filter for in-memory replication that potentially reduces network traffic by determining if session data needs to be replicated. In addition the `org.apache.catalina.ha.session.JvmRouteBinderValve` is enabled to ensure sessions are routed to a working node after the failure of a member (this Valve works in cooperation with `org.apache.catalina.ha.session.JVMRouteSessionIdBinderListener`).
- **`Deployer`** specifies the Java class (`org.apache.catalina.ha.deploy.FarmWarDeployer`) that deploys and undeploys distributed Web application WAR files from directories established in the `/tmp` folder.
- **`ClusterListener`** defines a listener for the cluster. The listener `org.apache.catalina.ha.session.JVMRouteSessionIdBinderListener` must be defined to assist `JvmRouteBinderValve` (see description of `JvmRouteBinderValve` above)

in routing a session to a cluster member after failure of the cluster member that originally handled the session.

You can customize your own configuration using the elements presented above. Refer to the Tomcat User Guide at `http://tomcat.apache.org/tomcat-10.0-doc/cluster-howto.html` (`https://tomcat.apache.org/tomcat-10.0-doc/cluster-howto.html`).

Web applications must have the `<distributable/>` element present in `web.xml` in order to participate in session replication. In addition, session objects must implement `java.io.Serializeable`. This marks a class as I/O capable.

11.2. Running Multiple Instances of Tomcat

❖ 11.2.1. Directory Setup

A separate configuration folder should be established for each Tomcat instance:

- Create a folder folder `/Tomcat10A`.
- Copy the `conf`, `logs`, `temp`, and `work` directories under `/Tomcat10` to `Tomcat10A`.
- All instances will use the same "webapps" folder and therefore will access the same web applications deployed to this folder. In `/Tomcat10A/server.xml` change the `webapps` attribute on the `Host` element to `/Tomcat10/webapps`. Save your changes.
- Create one additional folder `/Tomcat10B`. Copy the subfolders of `/Tomcat10A` to `/Tomcat10B`. Now you have two separate Tomcat directories and each directory has its own configuration files.

Each instance may be assigned a unique JVM route name so that a front end proxy (e.g., Apache) can associate an incoming request with the JVM of a particular instance. To accomplish this task, edit `/Tomcat10A/server.xml` and locate the `Engine` element.

Add the "jvmRoute" attribute as shown below:

```
<Engine name="Catalina" defaultHost="localhost" jvmRoute="Tomcat10A">
```

Repeat this edit for the `server.xml` file under the "Tomcat10B" folder. Specify a JVM route of "Tomcat10B".

❖ 11.2.2. Port number modifications

The following elements must have the `port` attribute changed to a unique value in `server.xml` under each folder:

- `Server` specifies shutdown port. For example, change the shutdown port to 8001, 8002 for Tomcat10A and Tomcat10B respectively.
- HTTP `Connector` specifies HTTP port. For example, change the HTTP port to 8181, 8282 for Tomcat10A and Tomcat10B respectively.
- AJP 1.3 `Connector` specifies AJP port (for communication between Apache and the clustered nodes). For example, change the AJP port to 8009, 8010 for Tomcat10A and Tomcat10B.
- Save your changes.

❖ 11.2.3. All to All with DeltaManager

The default cluster configuration specifies an "All to All" replication strategy implemented by **`DeltaManager`** . The `DeltaManager` broadcasts session data from one instance to the all other cluster members. This is acceptable for our purposes here, but the **`BackupManager`** is a more viable production alternative (discussed next).

❖ 11.2.4. Backup to One Cluster with BackupManager

With the `BackupManager`, session replication data is directed to one instance. To enable `BackupManager`, add a `Manager` element specifying a `className` of `org.apache.catalina.ha.session.BackupManager`. Add this element to `server.xml` as a child of the `Cluster` element. The advantage of using `BackupManager` is one cluster node stores session replication data, reducing traffic over the network as compared to `DeltaManager`. The disadvantage is if that node fails, then session failover to this node is not possible.

11.3. Enabling Session Replication

Tomcat supports 3 persistence methods to store session data. The built-in implementation for clustering (`SimpleTcpCluster`) only provides in-memory replication.

❖ 11.3.1. Session Persistence Using Shared File System

In this persistence strategy, session data is stored on a shared file system. You must comment out the `Cluster` element of each instance to disable in-memory session replication. For each instance, a `context.xml` file must be provided as the following demo illustrates.

Demo 11.1: tomcat-clustering/Demos/FilePersistenceLinux.xml

```
1.   <Context>
2.    <Manager className="org.apache.catalina.session.PersistentManager" >
3.     <Store className="org.apache.catalina.session.FileStore"
4.      directory="/devShareDir/TomcatClustering"
5.    />
6.    </Manager>
7.   </Context>
```


Code Explanation

The `Context` element contains a `Manager` element. A `Store` element is presented within the `Manager` element. The `directory` element points to a shared directory on the file system. This serves as the destination for replicated session data.

❖ 11.3.2. Session Persistence Using Shared Database

In this persistence strategy, session data is stored on a shared database.

Demo 11.2: tomcat-clustering/Demos/JDBCPersistence.xml

```
1.   <Context>
2.    <Manager className="org.apache.catalina.session.PersistentManager" >
3.     <Store className="org.apache.catalina.session.JDBCStore"
4.     connectionURL="jdbc:mysql://localhost:3306/sessionDB?user=admin;password=admin ↵↵
              pw"
5.     driverName="com.mysql.jdbc.Driver"
6.     sessionIdCol="session_id"
7.     sessionValidCol="valid"
8.     sessionMaxInactiveCol="max_inactive"
9.     sessionLastAccessCol="last_access"
10.    sessionTable="session_table"
11.    sessionAppCol="session_app"
12.    sessionDataCol="session_data" />
13.   </Manager>
14.  </Context>
```

Code Explanation

The `Manager` element remains unchanged from the previous demo. The `Store` element is coded with the following attributes:

- **`className`** name of the Java class implementing JDBC persistence.
- **`connectionURL`** the URL required to connect to the database, in this case a MySQL database.
- **`driverName`** name of the Java class required to connect to the database.
- **`connectionURL`** the URL required to connect to the database manager, in this case MySQL.
- **`sessionTable`** the name of the database table that will be used to store session data.
- **`sessionIdCol`** the name of the column in the session table to store session ID.
- **`sessionAppCol`** the name of the column in the session table to store the Engine, Host and Context name.
- **`sessionDataCol`** the name of the column in the session table to store session data.
- **`sessionValidCol`** the name of the column in the session table to indicate if session is valid.
- **`sessionMaxInactiveCol`** the name of the column in the session table to store the session's maximum inactive (timeout) interval.
- **`sessionLastAccessCol`** the name of the column in the session table to store the last access time.

❖ 11.3.3. Session Persistence using Shared Database: Database Table

Demo 11.3: tomcat-clustering/Demos/JDBCPersistenceTable.sql

```
1.   CREATE TABLE session_table (session_id varchar(1000) not null
2.     primary key,
3.     valid char(1)  not null,
4.     max_inactive  int not null,
5.     last_access  bigint,
6.     session_app  varchar(1000),
7.     session_data mediumblob);
```

Code Explanation

This file contains the SQL necessary to create the session table referenced in the previous demo.

❖ 11.3.4. In-memory Replication Using SimpleTcpCluster

In-memory replication of session information is implemented in the `SimpleTcpCluster` class provided by Tomcat.

11.4. Load Balancing Using mod_proxy Connector With Apache 2.4 Web Server

Load balancing is not provided by Tomcat 9. We look to the **Apache Web Server** to provide this capability.

The strategy for load balancing is to connect Apache to Tomca. Requests are directed to the Apache web server; Apache in turn will route these requests to one of the Tomcat instances defined in the Apache configuration. Communication is handled using the **Apache JServ Protocol** (AJP). The **`mod_proxy`** connector handles AJP on the Apache server and sends and receives AJP packets to and from Coyote (an AJP connector is available, and commented out, in `server.xml`).

Apache 2.4 ships with modules that provide load balancing capability: **`mod_proxy`**, **`mod_proxy_ajp`**, **`proxy_http_module`** and **`mod_proxy_balancer`**. These modules work together to achieve load balancing using Apache 2.4 as the front end proxy. For more information, visit the

mod_proxy_balancer documentation at https://httpd.apache.org/docs/2.4/mod/mod_proxy_balancer.html.

❖ 11.4.1. Using mod_proxy

The mod_proxy module supports directives that implement proxy functionality in Apache. This module is located in mod_proxy.so.

The mod_proxy module supports several protocols that enable Apache to talk to back-end servers. For example, HTTP communication is used by Apache to talk to any HTTP server (e.g., Tomcat, IIS, Websphere). AJP (Apache JServ Protocol) can be used to communicate with a Tomcat cluster. For a complete list of supported protocols see https://httpd.apache.org/docs/2.4/mod/mod_proxy.html.

The primary directive in this module is ProxyPass. This directive permits you to map a path to a remote server web resource address (URL). The Proxy container gives you the ability to specify directives for a URL defined in a ProxyPass directive such as security and load balancer requirements.

❖ 11.4.2. Forward vs. Reverse Proxy

Apache can serve as a forward proxy or as a reverse proxy.

To use Apache as a forward proxy, add the ProxyRequests on directive to your Apache configuration (this directive can also be coded in a virtual host container). You should also secure your server because a client can now anonymously access potentially any website. When using Apache as a forward proxy server you can secure Apache by using the Require directive in a Proxy container as illustrated below:

```
ProxyRequests on
    # www.safehost.com, www.goodguys.com are hypothetical trusted domains, domains that
 contain "phishers" are untrusted
    <Proxy "*">
        Require host www.safehost.com www.goodguys.com
        Require not host phishers
    </Proxy>
```

To use Apache as a reverse proxy, add one or more ProxyPass and ProxyPassReverse directives. A client using a reverse proxy is unaware of the back-end, or remote, server address. Apache maps a path, known to the client, to the remote site with the ProxyPass directive. For example, we wish to map "/tomcat" to the Tomcat home page, "http://localhost:8080":

```
ProxyPass /tomcat http://localhost:8080
```

The path (e.g., "/tomcat") is in effect a mirror of the remote site (e.g., "http://localhost:8080").

Note that the `ProxyPass` directive does not require `ProxyRequests on`.

The `ProxyPass` can also be specified in a `Location` container. In this case, the mapped path is coded on the location directive and is omitted on `ProxyPass`:

```
<Location /tomcat>
  ProxyPass http://localhost:8080
  Require host www.OurTomcatUsers.com
</Location>
```

If the back-end server sends a redirect request to the client, then the URL should be rewritten to the "local" URL. In this way, the browser sends a request that does not bypass the reverse proxy. To accomplish this task use the `ProxyPassReverse` directive in conjunction with `ProxyPass`:


```
ProxyPass /tomcat http://localhost:8080
ProxyPassReverse /tomcat http://localhost:8080
```

❖ 11.4.3. mod_proxy_balancer

Apache can also serve as a load balancer-a piece of software that directs traffic across a network of two or more servers. The routing of traffic can be based on server busyness (number of requests a server is assigned), pending requests or traffic (overall number of bytes a server is expected to handle).

In order to use the directives for load balancing, the `mod_proxy_balancer` module must be enabled in addition to `mod_proxy`.

A **virtual worker** must be defined to direct the request to a **real worker**. The real workers are specified as a list of `ProxyPass` directives contained within a `Proxy` container.

Consider the following example:

```
ProxyPass / balancer://myBalancer/ stickysession=JSESSIONID
<Proxy balancer://myBalancer >
  BalancerMember http://localhost:8181/ReplicateDemo/ loadfactor=10 route=Tomcat10A

  BalancerMember http://localhost:8282/ReplicateDemo/ loadfactor=5  route=Tomcat10B

   ProxySet lbmethod=byrequests
</Proxy>
```

In the example above, the `ProxyPass` directive associates the path "/" with a virtual worker. Note that the worker name is prefixed with "balancer"; this label indicates to Apache that a virtual worker is now defined for load balancing. The text following "balancer" is arbitrary. The "stickysession" attribute specifies that a session should be handled by the same real worker based on the `JSESSIONID` cookie name. Tomcat assigns the value of the session id to this cookie as well as the JVM route that has been assigned on the `Engine` element in `server.xml`. Apache compares the JVM route to the `route` name for each balancer member to determine where to direct the request.

The `Proxy` container defines the real workers. Each worker is associated with a back-end server, in this case a Tomcat cluster. Apache can communicate with each worker using either HTTP or AJP (Apache jServe Protocol). In our example we are using HTTP. Note that each worker must listen on a unique HTTP port.

The `loadfactor` weights each server for workload. The numeric values chosen are important in a relative sense. In the example above, a load factor of "10" indicates that the load balancing goal is to assign twice as much work to this worker as compared to the worker with an assigned load factor of "5". This attribute is meaningful if the `lbmethod` is assigned the value of `byrequests` or `bytraffic`.

The default value of `lbmethod` is `byrequests`. This value instructs Apache to balance the load based on the number of requests each server is currently processing. Assigning the load balancer the method of `bytraffic` is similar, except that overall byte count of the requests is also taken into consideration. The load balance method can also be specified as `bybusyness` which balances the load based on the number of requests queued for each server.

For a detailed discussion of `mod_proxy_balancer` go to `https://httpd.apache.org/docs/2.4/mod/mod_proxy_balancer.html`.

Exercise 13: Setting up Tomcat Clustering for High Availability

60 to 90 minutes

In this exercise, you will set up two Tomcat instances in a cluster to demonstrate high availability. High availability means that if one instance goes down, the other instance can respond to requests for web applications.

1. For this exercise you will use the Form Demo web application. The web application has already been deployed in earlier exercises.
2. Shut down the Tomcat server.
3. In the lesson demonstration you created two directories: `/Tomcat10A` and `/Tomcat10B`. Each directory contains a `server.xml` file with distinct port numbers for HTTP, server shut down and AJP. Take a few minutes to review each file.
4. Open `/Tomcat10/bin/setenv.sh` for edit. Comment out the line that sets the JMX properties for Catalina by inserting a "#" sign at the beginning of the line.
5. Add the following line after the comment:

```
export CATALINA_BASE=/Tomcat10A
```

6. Save your changes. Keep the edit session open as you will be making further chagnes to this file.
7. Start the Tomcat server. Tomcat will use the "Tomcat10A" configuration.
8. Next you will start the Apache web server as a proxy to the Tomcat server. A configuration file has already been provided so that Apache will pass requests for "FormDemo" to one of the Tomcat servers. In a terminal window, start Apache:

```
sudo httpd -k start
```

9. In your browser, type `http://localhost/FormDemo/` and hit **Enter**. The welcome page of the web application will be displayed. Enter your name and click the "submit" button to ensure the application is working correctly.

10. Return to the edit session on `/Tomcat10/bin/setenv.sh`. Change "Tomcat10A" to "Tomcat10B" as shown below:

```
export CATALINA_BASE=/Tomcat10B
```

11. Save your changes. Keep the edit session open as you will be making further changes to this file.
12. Start the Tomcat server. Tomcat will use the "Tomcat10B" configuration. You now have two Tomcat servers running.
13. In your browser, go to the web application: `http://localhost/FormDemo/` and hit **Enter**. The web application is operational.
14. Return to the edit session on `/Tomcat10/bin/setenv.sh`.
15. Change "Tomcat10B" to "Tomcat10A":

```
export CATALINA_BASE=/Tomcat10A
```

    Save your changes.

16. Shut down Tomcat. The "Tomcat10A" server is now stopped.
17. In your browser, test the "FormDemo" web application again. The web application is still operational because Apache has routed the request to "Tomcat10B".

Conclusion

In this lesson, you have learned

- The use of clustering for replication and load balancing
- How to set up clustering with front end proxy
- How to use the Apache web server for load balancing