Apache Tomcat

JNDI Datasource How-To

Introduction

JNDI Datasource configuration is covered extensively in the JNDI-Resources-HOWTO. However, feedback from tomcat-user has shown that specifics for individual configurations can be rather tricky.

Here then are some example configurations that have been posted to tomcat-user for popular databases and some general tips for db usage.

You should be aware that since these notes are derived from configuration and/or feedback posted to tomcat-user YMMV :-). Please let us know if you have any other tested configurations that you feel may be of use to the wider audience, or if you feel we can improve this section in anyway.

Please note that JNDI resource configuration changed somewhat between Tomcat 7.x and Tomcat 8.x as they are using different versions of Apache Commons DBCP library. You will most likely need to modify older JNDI resource configurations to match the syntax in the example below in order to make them work in Tomcat 11. See Tomcat Migration Guide for details.

Also, please note that JNDI DataSource configuration in general, and this tutorial in particular, assumes that you have read and understood the Context and Host configuration references, including the section about Automatic Application Deployment in the latter reference.

DriverManager, the service provider mechanism and memory leaks

java.sql.DriverManager supports the service provider mechanism. This feature is that all the available JDBC drivers that announce themselves by providing a META-INF/services/java.sql.Driver file are automatically discovered, loaded and registered, relieving you from the need to load the database driver explicitly before you create a JDBC connection. However, the implementation is fundamentally broken in all Java versions for a servlet container environment.

The problem is that java.sql.DriverManager will scan for the drivers only once.

The JRE Memory Leak Prevention Listener that is included with Apache Tomcat solves this by triggering the driver scan during Tomcat startup. This is enabled by default. It means that only libraries visible to the common class loader and its parents will be scanned for database drivers. This include drivers in $CATALINA_HOME/lib, $CATALINA_BASE/lib, the class path and the module path. Drivers packaged in web applications (in WEB-INF/lib) and in the shared class loader (where configured) will not be visible and will not be loaded automatically. If you are considering disabling this feature, note that the scan would be triggered by the first web application that is using JDBC, leading to failures when this web application is reloaded and for other web applications that rely on this feature.

Thus, the web applications that have database drivers in their WEB-INF/lib directory cannot rely on the service provider mechanism and should register the drivers explicitly.

The list of drivers in java.sql.DriverManager is also a known source of memory leaks. Any Drivers registered by a web application must be deregistered when the web application stops. Tomcat will attempt to automatically discover and deregister any JDBC drivers loaded by the web application class loader when the web application stops. However, it is expected that applications do this for themselves via a ServletContextListener.

Database Connection Pool (DBCP 2) Configurations

The default database connection pool implementation in Apache Tomcat relies on the libraries from the Apache Commons project. The following libraries are used:

- Commons DBCP 2
- Commons Pool 2

These libraries are located in a single JAR at $CATALINA_HOME/lib/tomcat-dbcp.jar. However, only the classes needed for connection pooling have been included, and the packages have been renamed to avoid interfering with

applications.

DBCP 2 provides support for JDBC 4.1.

Installation

See the DBCP 2 documentation for a complete list of configuration parameters.

Preventing database connection pool leaks

A database connection pool creates and manages a pool of connections to a database. Recycling and reusing already existing connections to a database is more efficient than opening a new connection.

There is one problem with connection pooling. A web application has to explicitly close ResultSet's, Statement's, and Connection's. Failure of a web application to close these resources can result in them never being available again for reuse, a database connection pool "leak". This can eventually result in your web application database connections failing if there are no more available connections.

There is a solution to this problem. The Apache Commons DBCP 2 can be configured to track and recover these abandoned database connections. Not only can it recover them, but also generate a stack trace for the code which opened these resources and never closed them.

To configure a DBCP 2 DataSource so that abandoned database connections are removed and recycled, add one or both of the following attributes to the Resource configuration for your DBCP 2 DataSource:

removeAbandonedOnBorrow=true

removeAbandonedOnMaintenance=true

The default for both of these attributes is false. Note that removeAbandonedOnMaintenance has no effect unless pool maintenance is enabled by setting timeBetweenEvictionRunsMillis to a positive value. See the DBCP 2 documentation for full documentation on these attributes.

Use the removeAbandonedTimeout attribute to set the number of seconds a database connection has been idle before it is considered abandoned.

removeAbandonedTimeout="60"

The default timeout for removing abandoned connections is 300 seconds.

The logAbandoned attribute can be set to true if you want DBCP 2 to log a stack trace of the code which abandoned the database connection resources.

logAbandoned="true"

The default is false.

MySQL DBCP 2 Example

0. Introduction

Versions of MySQL and JDBC drivers that have been reported to work:

- MySQL 3.23.47, MySQL 3.23.47 using InnoDB,, MySQL 3.23.58, MySQL 4.0.1alpha
- Connector/J 3.0.11-stable (the official JDBC Driver)
- mm.mysql 2.0.14 (an old 3rd party JDBC Driver)

Before you proceed, don't forget to copy the JDBC Driver's jar into $CATALINA_HOME/lib.

1. MySQL configuration

Ensure that you follow these instructions as variations can cause problems.

Create a new test user, a new database and a single test table. Your MySQL user **must** have a password assigned. The driver will fail if you try to connect with an empty password.

mysql> GRANT ALL PRIVILEGES ON *.* TO javauser@localhost

    ->   IDENTIFIED BY 'javadude' WITH GRANT OPTION;

mysql> create database javatest;

mysql> use javatest;

mysql> create table testdata (

```
    ->      id int not null auto_increment primary key,
    ->      foo varchar(25),
    ->      bar int);
```

Note: the above user should be removed once testing is complete!

Next insert some test data into the testdata table.

```
mysql> insert into testdata values(null, 'hello', 12345);
Query OK, 1 row affected (0.00 sec)

mysql> select * from testdata;
+----+-------+-------+
| ID | FOO   | BAR   |
+----+-------+-------+
|  1 | hello | 12345 |
+----+-------+-------+
1 row in set (0.00 sec)

mysql>
```

2. Context configuration

Configure the JNDI DataSource in Tomcat by adding a declaration for your resource to your Context.

For example:

```
<Context>

    <!-- maxTotal: Maximum number of database connections in pool. Make sure you
```

configure your mysqld max_connections large enough to handle

all of your db connections. Set to -1 for no limit.

-->

<!-- maxIdle: Maximum number of idle database connections to retain in pool.

Set to -1 for no limit. See also the DBCP 2 documentation on this

and the minEvictableIdleTimeMillis configuration parameter.

-->

<!-- maxWaitMillis: Maximum time to wait for a database connection to become available

in ms, in this example 10 seconds. An Exception is thrown if

this timeout is exceeded. Set to -1 to wait indefinitely.

-->

<!-- username and password: MySQL username and password for database connections -->

<!-- driverClassName: Class name for the old mm.mysql JDBC driver is

org.gjt.mm.mysql.Driver - we recommend using Connector/J though.

Class name for the official MySQL Connector/J driver is com.mysql.jdbc.Driver.

-->

```
    <!-- url: The JDBC connection url for connecting to your MySQL database.
         -->

  <Resource name="jdbc/TestDB" auth="Container"
type="javax.sql.DataSource"
                maxTotal="100" maxIdle="30" maxWaitMillis="10000"
                username="javauser" password="javadude"
driverClassName="com.mysql.jdbc.Driver"
                url="jdbc:mysql://localhost:3306/javatest"/>

</Context>
```

3. web.xml configuration

Now create a WEB-INF/web.xml for this test application.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                      https://jakarta.ee/xml/ns/jakartaee/web-app_6_1.xsd"
  version="6.1">
  <description>MySQL Test App</description>
  <resource-ref>
      <description>DB Connection</description>
      <res-ref-name>jdbc/TestDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
```

```
    </resource-ref>
</web-app>
```

4. Test code

Now create a simple test.jsp page for use later.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<sql:query var="rs" dataSource="jdbc/TestDB">
select id, foo, bar from testdata
</sql:query>

<html>
    <head>
        <title>DB Test</title>
    </head>
    <body>

    <h2>Results</h2>

<c:forEach var="row" items="${rs.rows}">
        Foo ${row.foo}<br/>
        Bar ${row.bar}<br/>
</c:forEach>
```

```
  </body>
</html>
```

That JSP page makes use of JSTL's SQL and Core taglibs. You can get it from Apache Tomcat Taglibs - Standard Tag Library project — just make sure you get a 1.1.x or later release. Once you have JSTL, copy jstl.jar and standard.jar to your web app's WEB-INF/lib directory.

Finally deploy your web app into $CATALINA_BASE/webapps either as a warfile called DBTest.war or into a sub-directory called DBTest

Once deployed, point a browser at http://localhost:8080/DBTest/test.jsp to view the fruits of your hard work.

Oracle 8i, 9i & 10g

0. Introduction

Oracle requires minimal changes from the MySQL configuration except for the usual gotchas :-)

Drivers for older Oracle versions may be distributed as *.zip files rather than *.jar files. Tomcat will only use *.jar files installed in $CATALINA_HOME/lib. Therefore classes111.zip or classes12.zip will need to be renamed with a .jar extension. Since jarfiles are zipfiles, there is no need to unzip and jar these files - a simple rename will suffice.

For Oracle 9i onwards you should use oracle.jdbc.OracleDriver rather than oracle.jdbc.driver.OracleDriver as Oracle have stated that oracle.jdbc.driver.OracleDriver is deprecated and support for this driver class will be discontinued in the next major release.

1. Context configuration

In a similar manner to the mysql config above, you will need to define your Datasource in your Context. Here we define a Datasource called myoracle using the thin driver to connect as user scott, password tiger to the sid called mysid. (Note: with the thin driver this sid is not the same as the tnsname). The schema used will be the default schema for the user scott.

Use of the OCI driver should simply involve a changing thin to oci in the URL string.

```
<Resource name="jdbc/myoracle" auth="Container"
                type="javax.sql.DataSource"
driverClassName="oracle.jdbc.OracleDriver"
                url="jdbc:oracle:thin:@127.0.0.1:1521:mysid"
                username="scott" password="tiger" maxTotal="20"
maxIdle="10"
                maxWaitMillis="-1"/>
```

2. web.xml configuration

You should ensure that you respect the element ordering defined by the DTD when you create you applications web.xml file.

```
<resource-ref>
  <description>Oracle Datasource example</description>
  <res-ref-name>jdbc/myoracle</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

3. Code example

You can use the same example application as above (assuming you create the required DB instance, tables etc.) replacing the Datasource code with something like

```
Context initContext = new InitialContext();
Context envContext  = (Context)initContext.lookup("java:/comp/env");
DataSource ds = (DataSource)envContext.lookup("jdbc/myoracle");
Connection conn = ds.getConnection();
```

//etc.

PostgreSQL

0. Introduction

PostgreSQL is configured in a similar manner to Oracle.

1. Required files

Copy the Postgres JDBC jar to $CATALINA_HOME/lib. As with Oracle, the jars need to be in this directory in order for DBCP 2's Classloader to find them. This has to be done regardless of which configuration step you take next.

2. Resource configuration

You have two choices here: define a datasource that is shared across all Tomcat applications, or define a datasource specifically for one application.

2a. Shared resource configuration

Use this option if you wish to define a datasource that is shared across multiple Tomcat applications, or if you just prefer defining your datasource in this file.

This author has not had success here, although others have reported so. Clarification would be appreciated here.

<Resource name="jdbc/postgres" auth="Container"

type="javax.sql.DataSource"
driverClassName="org.postgresql.Driver"

url="jdbc:postgresql://127.0.0.1:5432/mydb"

username="myuser" password="mypasswd" maxTotal="20"
maxIdle="10" maxWaitMillis="-1"/>

2b. Application-specific resource configuration

Use this option if you wish to define a datasource specific to your application, not visible to other Tomcat applications. This method is less invasive to your Tomcat installation.

Create a resource definition for your Context. The Context element should look

something like the following.

<Context>

<Resource name="jdbc/postgres" auth="Container"

type="javax.sql.DataSource"
driverClassName="org.postgresql.Driver"

url="jdbc:postgresql://127.0.0.1:5432/mydb"

username="myuser" password="mypasswd" maxTotal="20"
maxIdle="10"

maxWaitMillis="-1"/>

</Context>

3. web.xml configuration

<resource-ref>

<description>postgreSQL Datasource example</description>

<res-ref-name>jdbc/postgres</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

4. Accessing the datasource

When accessing the datasource programmatically, remember to
prepend java:/comp/env to your JNDI lookup, as in the following snippet of code.
Note also that "jdbc/postgres" can be replaced with any value you prefer,
provided you change it in the above resource definition file as well.

InitialContext cxt = new InitialContext();

if (cxt == null) {

```
    throw new Exception("Uh oh -- no context!");
}

DataSource ds = (DataSource) cxt.lookup( "java:/comp/env/jdbc/postgres" );

if ( ds == null ) {
    throw new Exception("Data source not found!");
}
```

Non-DBCP Solutions

These solutions either utilise a single connection to the database (not recommended for anything other than testing!) or some other pooling technology.

Oracle 8i with OCI client

Introduction

Whilst not strictly addressing the creation of a JNDI DataSource using the OCI client, these notes can be combined with the Oracle and DBCP 2 solution above.

In order to use OCI driver, you should have an Oracle client installed. You should have installed Oracle8i(8.1.7) client from cd, and download the suitable JDBC/OCI driver(Oracle8i 8.1.7.1 JDBC/OCI Driver) from otn.oracle.com.

After renaming classes12.zip file to classes12.jar for Tomcat, copy it into $CATALINA_HOME/lib. You may also have to remove the javax.sql.* classes from this file depending upon the version of Tomcat and JDK you are using.

Putting it all together

Ensure that you have the ocijdbc8.dll or .so in your $PATH or LD_LIBRARY_PATH (possibly in $ORAHOME\bin) and also confirm that the native library can be loaded by a simple test program using System.loadLibrary("ocijdbc8");

You should next create a simple test servlet or JSP that has these **critical lines**:

DriverManager.registerDriver(new

oracle.jdbc.driver.OracleDriver());

conn =

DriverManager.getConnection("jdbc:oracle:oci8:@database","username","password");

where database is of the form host:port:SID Now if you try to access the URL of your test servlet/JSP and what you get is a ServletException with a root cause of java.lang.UnsatisfiedLinkError:get_env_handle.

First, the UnsatisfiedLinkError indicates that you have

- a mismatch between your JDBC classes file and your Oracle client version. The giveaway here is the message stating that a needed library file cannot be found. For example, you may be using a classes12.zip file from Oracle Version 8.1.6 with a Version 8.1.5 Oracle client. The classesXXX.zip file and Oracle client software versions must match.
- A $PATH, LD_LIBRARY_PATH problem.
- It has been reported that ignoring the driver you have downloaded from otn and using the classes12.zip file from the directory $ORAHOME\jdbc\lib will also work.

Next you may experience the error ORA-06401 NETCMN: invalid driver designator

The Oracle documentation says : "Cause: The login (connect) string contains an invalid driver designator. Action: Correct the string and re-submit." Change the database connect string (of the form host:port:SID) with this one: (description=(address=(host=myhost)(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))

Ed. Hmm, I don't think this is really needed if you sort out your TNSNames - but I'm not an Oracle DBA :-)

Common Problems

Here are some common problems encountered with a web application which uses a database and tips for how to solve them.

Intermittent Database Connection Failures

Tomcat runs within a JVM. The JVM periodically performs garbage collection (GC) to remove java objects which are no longer being used. When the JVM performs GC execution of code within Tomcat freezes. If the maximum time configured for establishment of a database connection is less than the amount of time garbage collection took you can get a database connection failure.

To collect data on how long garbage collection is taking add the -verbose:gc argument to your CATALINA_OPTS environment variable when starting Tomcat. When verbose gc is enabled your $CATALINA_BASE/logs/catalina.out log file will include data for every garbage collection including how long it took.

When your JVM is tuned correctly 99% of the time a GC will take less than one second. The remainder will only take a few seconds. Rarely, if ever should a GC take more than 10 seconds.

Make sure that the db connection timeout is set to 10-15 seconds. For DBCP 2 you set this using the parameter maxWaitMillis.

Random Connection Closed Exceptions

These can occur when one request gets a db connection from the connection pool and closes it twice. When using a connection pool, closing the connection just returns it to the pool for reuse by another request, it doesn't close the connection. And Tomcat uses multiple threads to handle concurrent requests. Here is an example of the sequence of events which could cause this error in Tomcat:

Request 1 running in Thread 1 gets a db connection.

Request 1 closes the db connection.

The JVM switches the running thread to Thread 2

Request 2 running in Thread 2 gets a db connection

(the same db connection just closed by Request 1).

The JVM switches the running thread back to Thread 1

Request 1 closes the db connection a second time in a finally block.

The JVM switches the running thread back to Thread 2

Request 2 Thread 2 tries to use the db connection but fails

because Request 1 closed it.

Here is an example of properly written code to use a database connection obtained from a connection pool:

```
Connection conn = null;
Statement stmt = null;    // Or PreparedStatement if needed
ResultSet rs = null;
try {
    conn = ... get connection from connection pool ...
    stmt = conn.createStatement("select ...");
    rs = stmt.executeQuery();
    ... iterate through the result set ...
    rs.close();
```

```
        rs = null;
        stmt.close();
        stmt = null;
        conn.close(); // Return to connection pool
        conn = null;    // Make sure we don't close it twice
    } catch (SQLException e) {
        ... deal with errors ...
    } finally {
        // Always make sure result sets and statements are closed,
        // and the connection is returned to the pool
        if (rs != null) {
            try { rs.close(); } catch (SQLException e) { ; }
            rs = null;
        }
        if (stmt != null) {
            try { stmt.close(); } catch (SQLException e) { ; }
            stmt = null;
        }
        if (conn != null) {
            try { conn.close(); } catch (SQLException e) { ; }
            conn = null;
        }
    }
```

Context versus GlobalNamingResources

Please note that although the above instructions place the JNDI declarations in a Context element, it is possible and sometimes desirable to place these declarations in the GlobalNamingResources section of the server configuration file. A resource placed in the GlobalNamingResources section will be shared among the Contexts of the server.

JNDI Resource Naming and Realm Interaction

In order to get Realms to work, the realm must refer to the datasource as defined in the <GlobalNamingResources> or <Context> section, not a datasource as renamed using <ResourceLink>.

Class Loader How-To

Overview

Like many server applications, Tomcat installs a variety of class loaders (that is, classes that implement java.lang.ClassLoader) to allow different portions of the container, and the web applications running on the container, to have access to different repositories of available classes and resources. This mechanism is used to provide the functionality defined in the Servlet Specification, version 2.4 — in particular, Sections 9.4 and 9.6.

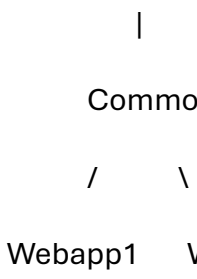
In a Java environment, class loaders are arranged in a parent-child tree. Normally, when a class loader is asked to load a particular class or resource, it delegates the request to a parent class loader first, and then looks in its own repositories only if the parent class loader(s) cannot find the requested class or resource. Note, that the model for web application class loaders *differs* slightly from this, as discussed below, but the main principles are the same.

When Tomcat is started, it creates a set of class loaders that are organized into the following parent-child relationships, where the parent class loader is above the child class loader:

Bootstrap

|

System



The characteristics of each of these class loaders, including the source of classes and resources that they make visible, are discussed in detail in the following section.

Class Loader Definitions

As indicated in the diagram above, Tomcat creates the following class loaders as it is initialized:

- **Bootstrap** — This class loader contains the basic runtime classes provided by the Java Virtual Machine, plus any classes from JAR files present in the System Extensions directory ($JAVA_HOME/jre/lib/ext). *Note*: some JVMs may implement this as more than one class loader, or it may not be visible (as a class loader) at all.
- **System** — This class loader is normally initialized from the contents of the CLASSPATH environment variable. All such classes are visible to both Tomcat internal classes, and to web applications. However, the standard Tomcat startup scripts ($CATALINA_HOME/bin/catalina.sh or %CATALINA_HOME%\bin\catalina.bat) totally ignore the contents of the CLASSPATH environment variable itself, and instead build the System class loader from the following repositories:
    - *$CATALINA_HOME/bin/bootstrap.jar* — Contains the main() method that is used to initialize the Tomcat server, and the class loader implementation classes it depends on.
    - *$CATALINA_BASE/bin/tomcat-juli.jar* or *$CATALINA_HOME/bin/tomcat-juli.jar* — Logging implementation classes. These include enhancement classes

to java.util.logging API, known as Tomcat JULI, and a package-renamed copy of Apache Commons Logging library used internally by Tomcat. See logging documentation for more details.

If tomcat-juli.jar is present in *$CATALINA_BASE/bin*, it is used instead of the one in *$CATALINA_HOME/bin*. It is useful in certain logging configurations

- *$CATALINA_HOME/bin/commons-daemon.jar* — The classes from Apache Commons Daemon project. This JAR file is not present in the CLASSPATH built by catalina.bat|.sh scripts, but is referenced from the manifest file of *bootstrap.jar*.

- **Common** — This class loader contains additional classes that are made visible to both Tomcat internal classes and to all web applications.

Normally, application classes should **NOT** be placed here. The locations searched by this class loader are defined by the common.loader property in $CATALINA_BASE/conf/catalina.properties. The default setting will search the following locations in the order they are listed:

- unpacked classes and resources in $CATALINA_BASE/lib
- JAR files in $CATALINA_BASE/lib
- unpacked classes and resources in $CATALINA_HOME/lib
- JAR files in $CATALINA_HOME/lib

By default, this includes the following:

- *annotations-api.jar* — Jakarta Annotations 3.0 classes.
- *catalina.jar* — Implementation of the Catalina servlet container portion of Tomcat.
- *catalina-ant.jar* — Optional. Tomcat Catalina Ant tasks for working with the Manager web application.
- *catalina-ha.jar* — Optional. High availability package that provides session clustering functionality built on Tribes.
- *catalina-ssi.jar* — Optional. Server-side Includes module.

- *catalina-storeconfig.jar* — Optional. Generation of XML configuration files from current state.
- *catalina-tribes.jar* — Optional. Group communication package used by the high availabaility package.
- *ecj-*.jar* — Optional. Eclipse JDT Java compiler used to compile JSPs to Servlets.
- *el-api.jar* — Optional. EL 6.0 API.
- *jakartaee-migration-*-shaded.jar* — Optional. Provides conversion of web applications from Java EE 8 to Jakarta EE 9.
- *jasper.jar* — Optional. Tomcat Jasper JSP Compiler and Runtime.
- *jasper-el.jar* — Optional. Tomcat EL implementation.
- *jaspic-api.jar* — Jakarta Authentication 3.1 API.
- *jsp-api.jar* — Optional. Jakarta Pages 4.0 API.
- *servlet-api.jar* — Jakarta Servlet 6.1 API.
- *tomcat-api.jar* — Several interfaces defined by Tomcat.
- *tomcat-coyote.jar* — Tomcat connectors and utility classes.
- *tomcat-dbcp.jar* — Optional. Database connection pool implementation based on package-renamed copy of Apache Commons Pool 2 and Apache Commons DBCP 2.
- *tomcat-i18n-**.jar* — Optional JARs containing resource bundles for other languages. As default bundles are also included in each individual JAR, they can be safely removed if no internationalization of messages is needed.
- *tomcat-jdbc.jar* — Optional. An alternative database connection pool implementation, known as Tomcat JDBC pool.
  See documentation for more details.
- *tomcat-jni.jar* — Provides the integration with the Tomcat Native

library.

    - *tomcat-util.jar* — Common classes used by various components of Apache Tomcat.
    - *tomcat-util-scan.jar* — Provides the class scanning functionality used by Tomcat.
    - *tomcat-websocket.jar* — Optional. Jakarta WebSocket 2.2 implementation
    - *websocket-api.jar* — Optional. Jakarta WebSocket 2.2 API
    - *websocket-client-api.jar* — Optional. Jakarta WebSocket 2.2 Client API
- **WebappX** — A class loader is created for each web application that is deployed in a single Tomcat instance. All unpacked classes and resources in the /WEB-INF/classes directory of your web application, plus classes and resources in JAR files under the /WEB-INF/lib directory of your web application, are made visible to this web application, but not to other ones.

As mentioned above, the web application class loader diverges from the default Java delegation model (in accordance with the recommendations in the Servlet Specification, version 2.4, section 9.7.2 Web Application Classloader). When a request to load a class from the web application's *WebappX* class loader is processed, this class loader will look in the local repositories **first**, instead of delegating before looking. There are exceptions. Classes which are part of the JRE base classes cannot be overridden. There are some exceptions such as the XML parser components which can be overridden using the upgradeable modules feature. Lastly, the web application class loader will always delegate first for Jakarta EE API classes for the specifications implemented by Tomcat (Servlet, JSP, EL, WebSocket). All other class loaders in Tomcat follow the usual delegation pattern.

Therefore, from the perspective of a web application, class or resource loading looks in the following repositories, in this order:

- Bootstrap classes of your JVM
- */WEB-INF/classes* of your web application
- */WEB-INF/lib/*.jar* of your web application
- System class loader classes (described above)
- Common class loader classes (described above)

If the web application class loader is configured with <Loader delegate="true"/> then the order becomes:

- Bootstrap classes of your JVM
- System class loader classes (described above)
- Common class loader classes (described above)
- */WEB-INF/classes* of your web application
- */WEB-INF/lib/*.jar* of your web application

XML Parsers and Java

In older versions of Tomcat, you could simply replace the XML parser in the Tomcat libraries directory to change the parser used by all web applications. However, this technique will not be effective when you are running modern versions of Java, because the usual class loader delegation process will always choose the implementation inside the JDK in preference to this one.

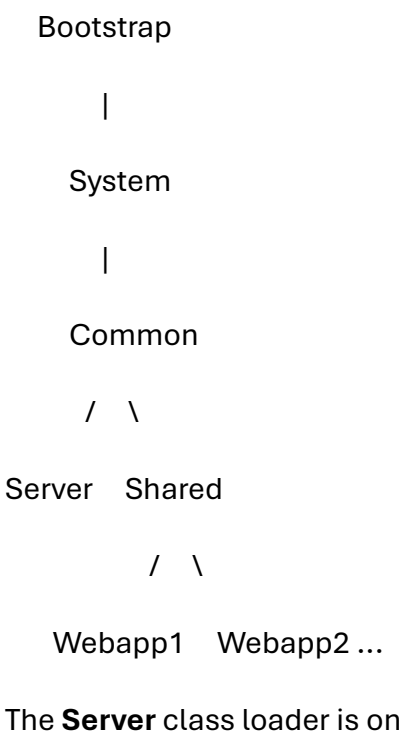
Java supports a mechanism called upgradeable modules to allow replacement of APIs created outside of the JCP (i.e. DOM and SAX from W3C). It can also be used to update the XML parser implementation.

Note that overriding any JRE component carries risk. If the overriding component does not provide a 100% compatible API (e.g. the API provided by Xerces is not 100% compatible with the XML API provided by the JRE) then there is a risk that Tomcat and/or the deployed application will experience errors.

Advanced configuration

A more complex class loader hierarchy may also be configured. See the diagram

below. By default, the **Server** and **Shared** class loaders are not defined and the simplified hierarchy shown above is used. This more complex hierarchy may be use by defining values for the server.loader and/or shared.loader properties in conf/catalina.properties.



The **Server** class loader is only visible to Tomcat internals and is completely invisible to web applications.

The **Shared** class loader is visible to all web applications and may be used to shared code across all web applications. However, any updates to this shared code will require a Tomcat restart.

Jasper 2 JSP Engine How To

Introduction

Tomcat 11.0 uses the Jasper 2 JSP Engine to implement the Jakarta Pages 4.0 specification.

Jasper 2 has been redesigned to significantly improve performance over the original Jasper. In addition to general code improvements the following changes were made:

- **JSP Custom Tag Pooling** - The java objects instantiated for JSP Custom Tags can now be pooled and reused. This significantly boosts the performance of JSP pages which use custom tags.
- **Background JSP compilation** - If you make a change to a JSP page which had already been compiled Jasper 2 can recompile that page in the background. The previously compiled JSP page will still be available to serve requests. Once the new page has been compiled successfully it will replace the old page. This helps improve availability of your JSP pages on a production server.
- **Recompile JSP when included page changes** - Jasper 2 can now detect when a page included at compile time from a JSP has changed and then recompile the parent JSP.
- **JDT used to compile JSP pages** - The Eclipse JDT Java compiler is now used to perform JSP java source code compilation. This compiler loads source dependencies from the container classloader. Ant and javac can still be used.

Jasper is implemented using the servlet class org.apache.jasper.servlet.JspServlet.

Configuration

By default Jasper is configured for use when doing web application development. See the section Production Configuration for information on configuring Jasper for use on a production Tomcat server.

The servlet which implements Jasper is configured using init parameters in your global $CATALINA_BASE/conf/web.xml.

- **checkInterval** - If development is false and checkInterval is greater than zero, background compiles are enabled. checkInterval is the time in seconds between checks to see if a JSP page (and its dependent files) needs to be recompiled. Default 0 seconds.
- **classdebuginfo** - Should the class file be compiled with debugging information? true or false, default true.
- **classpath** - Defines the class path to be used to compile the generated servlets. This parameter only has an effect if the ServletContext attribute org.apache.jasper.Constants.SERVLET_CLASSPATH is not set. This attribute is always set when Jasper is used within Tomcat. By default the classpath is created dynamically based on the current web application.
- **compiler** - Which compiler Ant should use to compile JSP pages. The valid values for this are the same as for the compiler attribute of Ant's javac task. If the value is not set, then the default Eclipse JDT Java compiler will be used instead of using Ant. There is no default value. If this attribute is set then setenv.[sh|bat] should be used to add ant.jar, ant-launcher.jar and tools.jar to the CLASSPATH environment variable.
- **compilerSourceVM** - What JDK version are the source files compatible with? (Default value: 17)
- **compilerTargetVM** - What JDK version are the generated files compatible with? (Default value: 17)
- **development** - Is Jasper used in development mode? If true, the frequency at which JSPs are checked for modification may be specified via the modificationTestInterval parameter.true or false, default true.
- **displaySourceFragment** - Should a source fragment be included in exception messages? true or false, default true.
- **dumpSmap** - Should the SMAP info for JSR45 debugging be dumped to a file? true or false, default false. false if suppressSmap is true.

- **enablePooling** - Determines whether tag handler pooling is enabled. This is a compilation option. It will not alter the behaviour of JSPs that have already been compiled. true or false, default true.
- **engineOptionsClass** - Allows specifying the Options class used to configure Jasper. If not present, the default EmbeddedServletOptions will be used.
- **errorOnUseBeanInvalidClassAttribute** - Should Jasper issue an error when the value of the class attribute in an useBean action is not a valid bean class? true or false, default true.
- **fork** - Have Ant fork JSP page compiles so they are performed in a separate JVM from Tomcat? true or false, default true.
- **genStringAsCharArray** - Should text strings be generated as char arrays, to improve performance in some cases? Default false.
- **javaEncoding** - Java file encoding to use for generating java source files. Default UTF8.
- **keepgenerated** - Should we keep the generated Java source code for each page instead of deleting it? true or false, default true.
- **mappedfile** - Should we generate static content with one print statement per input line, to ease debugging? true or false, default true.
- **maxLoadedJsps** - The maximum number of JSPs that will be loaded for a web application. If more than this number of JSPs are loaded, the least recently used JSPs will be unloaded so that the number of JSPs loaded at any one time does not exceed this limit. A value of zero or less indicates no limit. Default -1
- **jspIdleTimeout** - The amount of time in seconds a JSP can be idle before it is unloaded. A value of zero or less indicates never unload. Default -1
- **modificationTestInterval** - Causes a JSP (and its dependent files) to not be checked for modification during the specified time interval (in seconds) from the last time the JSP was checked for modification. A value

of 0 will cause the JSP to be checked on every access. Used in development mode only. Default is 4 seconds.

- **recompileOnFail** - If a JSP compilation fails should the modificationTestInterval be ignored and the next access trigger a re-compilation attempt? Used in development mode only and is disabled by default as compilation may be expensive and could lead to excessive resource usage.
- **scratchdir** - What scratch directory should we use when compiling JSP pages? Default is the work directory for the current web application.
- **suppressSmap** - Should the generation of SMAP info for JSR45 debugging be suppressed? true or false, default false.
- **trimSpaces** - Should template text that consists entirely of whitespace be removed from the output (true), replaced with a single space (single) or left unchanged (false)? Alternatively, the extended option will remove leading and trailing whitespace from template text and collapse sequences of whitespace and newlines within the template text to a single new line. Note that if a JSP page or tag file specifies a trimDirectiveWhitespaces value of true, that will take precedence over this configuration setting for that page/tag. Default false.
- **xpoweredBy** - Determines whether X-Powered-By response header is added by generated servlet. true or false, default false.
- **strictQuoteEscaping** - When scriptlet expressions are used for attribute values, should the rules in JSP.1.6 for the escaping of quote characters be strictly applied? true or false, default true.
- **quoteAttributeEL** - When EL is used in an attribute value on a JSP page, should the rules for quoting of attributes described in JSP.1.6 be applied to the expression? true or false, default true.
- **variableForExpressionFactory** - The name of the variable to use for the expression language expression factory. If not specified, the default value of _el_expressionfactory will be used.

- **variableForInstanceManager** - The name of the variable to use for the instance manager factory. If not specified, the default value of _jsp_instancemanager will be used.
- **poolTagsWithExtends** - By default, JSPs that use their own base class via the extends attribute of the page directive, will have Tag pooling disabled since Jasper cannot guarantee that the necessary initialisation will have taken place. This can have a negative impact on performance. Providing the alternative base class calls _jspInit() from Servlet.init(), setting this property to true will enable pooling with an alternative base class. If the alternative base class does not call _jspInit() and this property is true, NPEs will occur when attempting to use tags. true or false, default false.
- **strictGetProperty** - If true, the requirement to have the object referenced in jsp:getProperty action to be previously "introduced" to the JSP processor, as specified in the chapter JSP.5.3 of JSP 2.0 and later specifications, is enforced. true or false, default true.
- **strictWhitespace** - If false the requirements for whitespace before an attribute name will be relaxed so that the lack of whitespace will not cause an error. true or false, default true.
- **jspServletBase** - The base class of the Servlets generated from the JSPs.If not specified, the default value of org.apache.jasper.runtime.HttpJspBase will be used.
- **serviceMethodName** - The name of the service method called by the base class. If not specified, the default value of _jspService will be used.
- **servletClasspathAttribute** - The name of the ServletContext attribute that provides the classpath for the JSP. If not specified, the default value of org.apache.catalina.jsp_classpath will be used.
- **jspPrecompilationQueryParameter** - The name of the query parameter that causes the JSP engine to just pregenerate the servlet but not invoke it. If not specified, the default value of jsp_precompile will be used, as defined by JSP specification (JSP.11.4.2).

- **generatedJspPackageName** - The default package name for compiled JSPs. If not specified, the default value of org.apache.jsp will be used.
- **generatedTagFilePackageName** - The default package name for tag handlers generated from tag files. If not specified, the default value of org.apache.jsp.tag will be used.
- **tempVariableNamePrefix** - Prefix to use for generated temporary variable names. If not specified, the default value of _jspx_temp will be used.
- **useInstanceManagerForTags** - If true, the instance manager is used to obtain tag handler instances. true or false, default false.
- **limitBodyContentBuffer** - If true, any tag buffer that expands beyond the value of the bodyContentTagBufferSize init parameter will be destroyed and a new buffer created. true or false, default false.
- **bodyContentTagBufferSize** - The size (in characters) to use when creating a tag buffer. If not specified, the default value of org.apache.jasper.Constants.DEFAULT_TAG_BUFFER_SIZE (512) will be used.

The Java compiler from Eclipse JDT in included as the default compiler. It is an advanced Java compiler which will load all dependencies from the Tomcat class loader, which will help tremendously when compiling on large installations with tens of JARs. On fast servers, this will allow sub-second recompilation cycles for even large JSP pages.

Apache Ant, which was used in previous Tomcat releases, can be used instead of the new compiler by configuring the compiler attribute as explained above.

If you need to change the JSP Servlet settings for an application you can override the default configuration by re-defining the JSP Servlet in /WEB-INF/web.xml. However, this may cause problems if you attempt to deploy the application on another container as the JSP Servlet class may not be recognised. You can work-around this problem by using the Tomcat specific /WEB-INF/tomcat-web.xml deployment descriptor. The format is identical to /WEB-INF/web.xml. It will override any default settings but not those in /WEB-INF/web.xml. Since it is

Tomcat specific, it will only be processed when the application is deployed on Tomcat.

Known issues

As described in bug 39089, a known JVM issue, bug 6294277, may cause a java.lang.InternalError: name is too long to represent exception when compiling very large JSPs. If this is observed then it may be worked around by using one of the following:

- reduce the size of the JSP
- disable SMAP generation and JSR-045 support by setting suppressSmap to true.

Production Configuration

The main JSP optimization which can be done is precompilation of JSPs. However, this might not be possible (for example, when using the jsp-property-group feature) or practical, in which case the configuration of the Jasper servlet becomes critical.

When using Jasper 2 in a production Tomcat server you should consider making the following changes from the default configuration.

- **development** - To disable on access checks for JSP pages compilation set this to false.
- **genStringAsCharArray** - To generate slightly more efficient char arrays, set this to true.
- **modificationTestInterval** - If development has to be set to true for any reason (such as dynamic generation of JSPs), setting this to a high value will improve performance a lot.
- **trimSpaces** - To remove unnecessary bytes from the response, consider setting this to single or extended.

Web Application Compilation

Using Ant is the preferred way to compile web applications using JSPC. Note that

when pre-compiling JSPs, SMAP information will only be included in the final classes if suppressSmap is false and compile is true. Use the script given below (a similar script is included in the "deployer" download) to precompile a webapp:

```
<project name="Webapp Precompilation" default="all" basedir=".">

    <import file="${tomcat.home}/bin/catalina-tasks.xml"/>

    <target name="jspc">

      <jasper
                    validateXml="false"
                    uriroot="${webapp.path}"
                    webXmlInclude="${webapp.path}/WEB-INF/generated_web.xml"
                    outputDir="${webapp.path}/WEB-INF/src" />

  </target>

  <target name="compile">

      <mkdir dir="${webapp.path}/WEB-INF/classes"/>
      <mkdir dir="${webapp.path}/WEB-INF/lib"/>

      <javac destdir="${webapp.path}/WEB-INF/classes"
                debug="on" failonerror="false"
                srcdir="${webapp.path}/WEB-INF/src"
```

```
            excludes="**/*.smap">
      <classpath>
        <pathelement location="${webapp.path}/WEB-INF/classes"/>
        <fileset dir="${webapp.path}/WEB-INF/lib">
          <include name="*.jar"/>
        </fileset>
        <pathelement location="${tomcat.home}/lib"/>
        <fileset dir="${tomcat.home}/lib">
          <include name="*.jar"/>
        </fileset>
        <fileset dir="${tomcat.home}/bin">
          <include name="*.jar"/>
        </fileset>
      </classpath>
      <include name="**" />
      <exclude name="tags/**" />
    </javac>

</target>

<target name="all" depends="jspc,compile">
</target>

<target name="cleanup">
```

```
    <delete>
        <fileset dir="${webapp.path}/WEB-INF/src"/>
        <fileset dir="${webapp.path}/WEB-INF/classes/org/apache/jsp"/>
    </delete>
  </target>


</project>
```

The following command line can be used to run the script (replacing the tokens with the Tomcat base path and the path to the webapp which should be precompiled):

$ANT_HOME/bin/ant -Dtomcat.home=<$TOMCAT_HOME> -Dwebapp.path=<$WEBAPP_PATH>

Then, the declarations and mappings for the servlets which were generated during the precompilation must be added to the web application deployment descriptor. Insert the ${webapp.path}/WEB-INF/generated_web.xml at the right place inside the ${webapp.path}/WEB-INF/web.xml file. Restart the web application (using the manager) and test it to verify it is running fine with precompiled servlets. An appropriate token placed in the web application deployment descriptor may also be used to automatically insert the generated servlet declarations and mappings using Ant filtering capabilities. This is actually how all the webapps distributed with Tomcat are automatically compiled as part of the build process.

At the jasper task you can use the option addWebXmlMappings for automatic merge the ${webapp.path}/WEB-INF/generated_web.xml with the current web application deployment descriptor at ${webapp.path}/WEB-INF/web.xml.

When you want to use a specific version of Java for your JSP's, add the javac compiler task attributes source and target with appropriate values. For example, 16 to compile JSPs for Java 16.

For production you may wish to disable debug info with debug="off".

When you don't want to stop the JSP generation at first JSP syntax error, use failOnError="false" and with showSuccess="true" all successful *JSP to Java* generation are printed out. Sometimes it is very helpful, when you cleanup the generate java source files at ${webapp.path}/WEB-INF/src and the compile JSP servlet classes at ${webapp.path}/WEB-INF/classes/org/apache/jsp.

Hints:

- When you switch to another Tomcat release, then regenerate and recompile your JSP's with the new Tomcat version.
- Use a Servlet context parameter to disable PageContext pooling org.apache.jasper.runtime.JspFactoryImpl.POOL_SIZE=-1 and limit the buffering with the JSP Servlet init-param limitBodyContentBuffer=true. Note that changing from the defaults may affect performance, but it will vary depending on the application.

Optimisation

There are a number of extension points provided within Jasper that enable the user to optimise the behaviour for their environment.

The first of these extension points is the tag plug-in mechanism. This allows alternative implementations of tag handlers to be provided for a web application to use. Tag plug-ins are registered via a tagPlugins.xml file located under WEB-INF. A sample plug-in for the JSTL is included with Jasper.

The second extension point is the Expression Language interpreter. Alternative interpreters may be configured through the ServletContext. See the ELInterpreterFactory javadoc for details of how to configure an alternative EL interpreter. A alternative interpreter primarily targeting tag settings is provided at org.apache.jasper.optimizations.ELInterpreterTagSetters. See the javadoc for details of the optimisations and the impact they have on specification compliance.

An extension point is also provided for coercion of String values to Enums. It is provided at org.apache.jasper.optimizations.StringInterpreterEnum. See the

javadoc for details of the optimisations and the impact they have on specification compliance.