FREE eBook

LEARNING tomcat

Free unaffiliated eBook created from
Stack Overflow contributors.

#tomcat

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: tomcat

It is an unofficial and free tomcat ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official tomcat.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with tomcat

Remarks

This section provides an overview of what tomcat is, and why a developer might want to use it.

It should also mention any large subjects within tomcat, and link out to the related topics. Since the Documentation for tomcat is new, you may need to create initial versions of those related topics.

Versions

Version	Java	Servlet	JSP	EL	WebSocket	JASPIC	Released
6.0.x	5+	2.5	2.1	2.1	n/a	n/a	2006-12-01
7.0.x	6+	3.0	2.2	2.2	1.1	n/a	2010-06-02
8.0.x	7+	3.1	2.3	3.0	1.1	n/a	2013-08-05
8.5.x	7+	3.1	2.3	3.0	1.1	1.1	2016-06-13
9.0.x	8+	4.0	2.4	3.1	1.2	1.1	2016-06-13

Examples

Installation or Setup

Detailed instructions on getting tomcat set up or installed.

Installing Tomcat as a service on Ubuntu

This example demonstrates how to install Tomcat as a service on Ubuntu using the *.*tar.gz* releases of both Tomcat as well as Java.

1. Install the Java Runtime Environment (JRE)

1. Download the desired jre *.tar.gz* release
2. Extract to `/opt/`
   This will create a directory `/opt/jre1.Xxxx/`
3. Create a symbolic link to the java home directory:
   ```
   cd /opt; sudo ln -s jre1.Xxxxx java
   ```
4. add the JRE to the JAVA_HOME environment variable:
   ```
   sudo vim /etc/environment
   JAVA_HOME="/opt/java"
   ```

2. Install Tomcat:

1. Download tomcat in a *.tar.gz* (or similiar) release.
2. Create a tomcat system user:
   `sudo useradd -r tomcat`
3. Extract to `/opt/`
   This will create a directory `/opt/apache-tomcat-XXXX`
   assign this directory to the tomcat system user and group:
   `sudo chown -R tomcat ./*`
   `sudo chgrp -R tomcat ./*`
4. Create the `CATALINA_HOME` environment variable:
   `sudo vim /etc/environment`
   `CATALINA_HOME="/opt/tomcat"`
5. Add admin user in `tomcat-users.xml`
   `sudo vim /opt/tomcat/conf/tomcat-users.xml`
   and add something like `&ltuser username="admin" password="adminpw" roles="manager-gui">`
   between the `&lttomcat-users>` ... `</tomcat-users>` tags

3. Making Tomcat boot at startup

Add a script in /etc/init.d called tomcat and make it executable. The content of the script can look something like:

```
RETVAL=$?
CATALINA_HOME="/opt/tomcat"

case "$1" in
 start)
    if [ -f $CATALINA_HOME/bin/startup.sh ];
      then
        echo $"Starting Tomcat"
        sudo -u tomcat $CATALINA_HOME/bin/startup.sh
    fi
    ;;
 stop)
    if [ -f $CATALINA_HOME/bin/shutdown.sh ];
      then
        echo $"Stopping Tomcat"
        sudo -u tomcat $CATALINA_HOME/bin/shutdown.sh
    fi
    ;;
 *)
    echo $"Usage: $0 {start|stop}"
    exit 1
    ;;
esac

exit $RETVAL
```

To make it start on boot, run: `sudo update-rc.d tomcat defaults`

You can also add a bash line to /etc/rc.local for example `service tomcat start`

Changing classpath or other Tomcat related environment variables:

Edit the file `$CATALINA_HOME/bin/setenv.sh` and add the properties in here, for example:
`CLASSPATH=/additional/class/directories`

Read Getting started with tomcat online: https://riptutorial.com/tomcat/topic/2107/getting-started-with-tomcat

Chapter 2: CAC enabling Tomcat for Development Purposes

Examples

Creating the Keystores and configuring Tomcat

This writeup walks though steps to configure Tomcat to request CAC certificates from the client. It is focused on setting up a development environment, so some features that should be considered for production are not here. (For example it shows using a self-signed certificate for https and it doesn't consider checking for revoked certificates.)

Create Keystore for enabling HTTPS connections

The first step is to set up SSL on tomcat. This is documented on the tomcat website here: https://tomcat.apache.org/tomcat-8.5-doc/ssl-howto.html for completeness the steps to set it up with a self-signed certificate are listed below:

We need to create a keystore file that holds the SSL certificate for the server. The certificate is what is required to create an https connection and doesn't have anything to do with making the server request CAC certificates from the client but https connections are required for client certificate authentication. For a development environment creating a self-signed certificate is ok but it's discouraged for production. Java comes packaged with a utility called keytool (http://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html) that is used to managed certificates and keystores. It can be used to create a self signed certificate and add it to a keystore. To do that you can issue the following command from a command prompt:

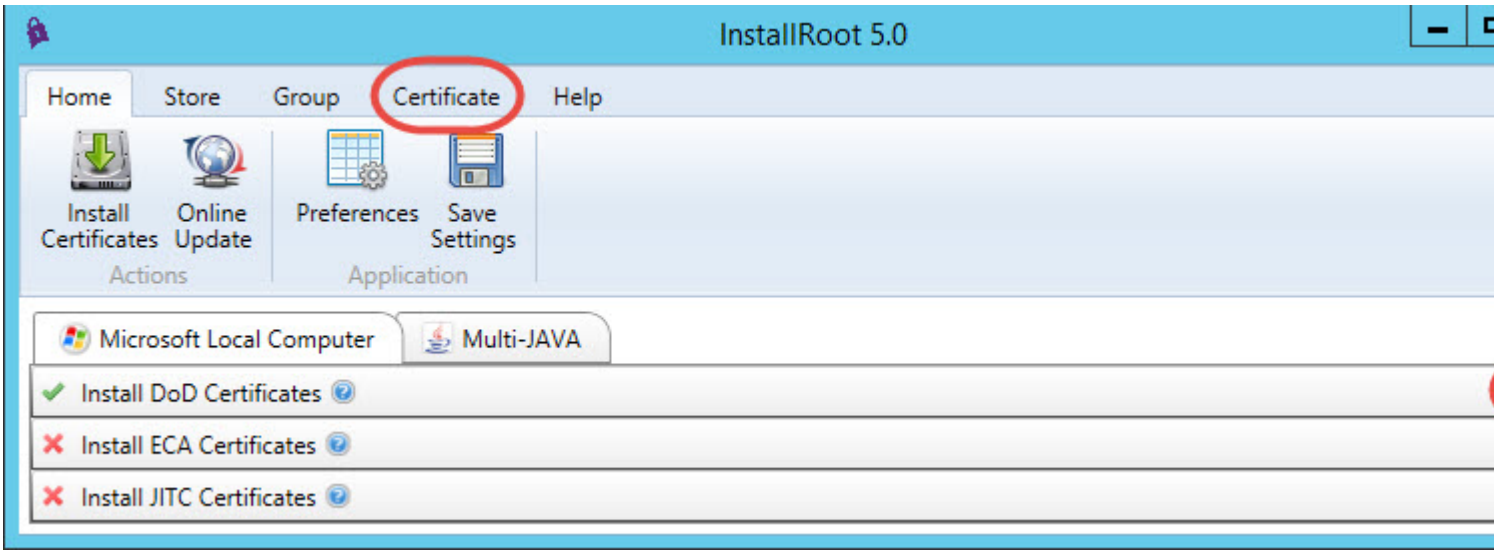
> keytool -genkey -alias tomcat -keyalg RSA -keystore \path\to\my\keystore -storepass changeit

You will be prompted for various bits of information and then a keystore file named "\path\to\my\keystore" with a password of 'changeit' will be created and it will contain the generate self-signed certificate.

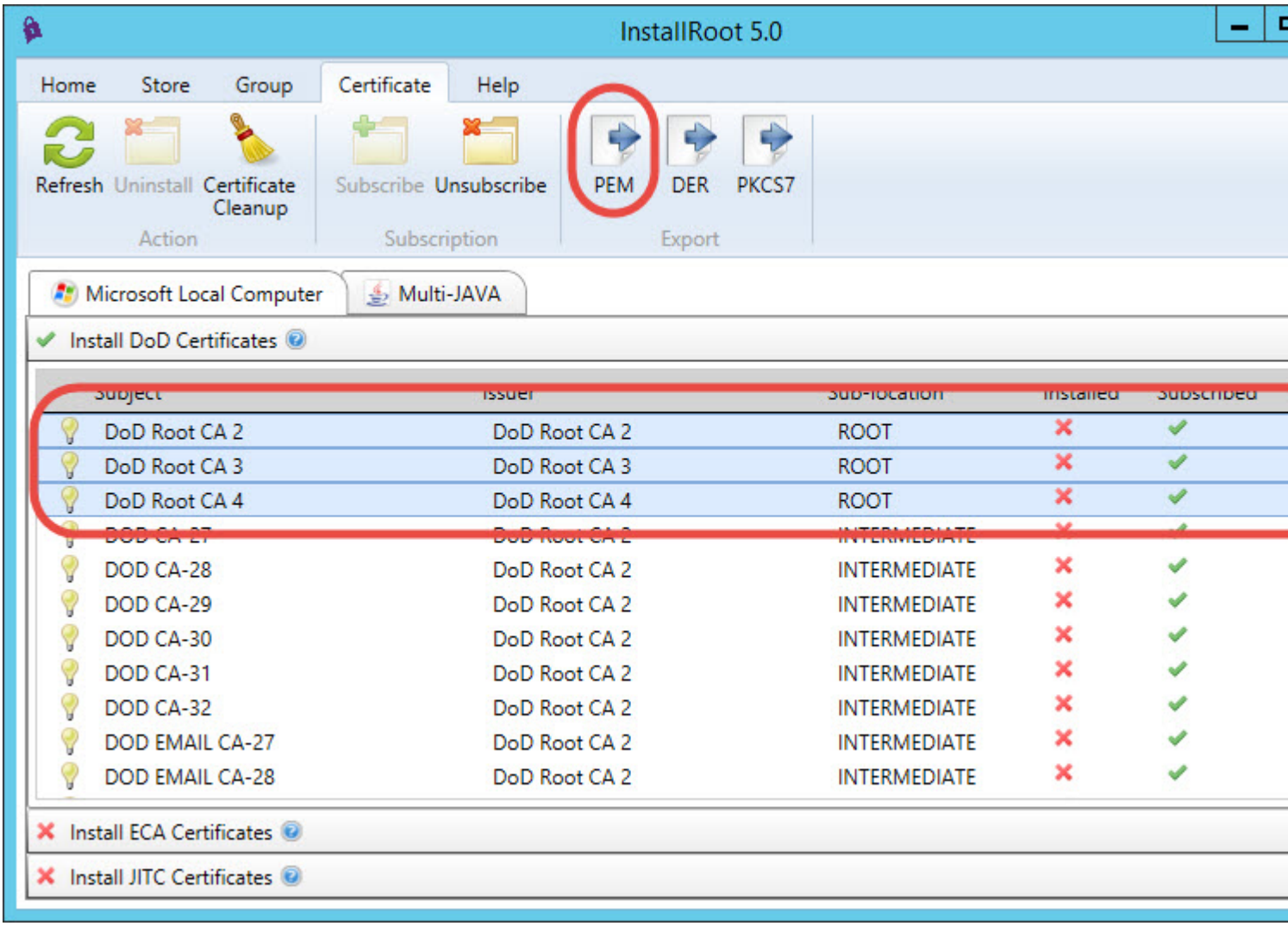
Create truststore containing DoD root certificates

The next thing that is needed is to create a truststore that will contain the DoD root certificates. The certificates in this truststore will be considered as trusted by tomcat and it will only accept client certificates that have one of the trusted certs in their certificate chain.

To create the truststore we need to get a copy of the DoD root certificates. To do this download "InstallRoot 5.0" from http://militarycac.com/dodcerts.htm. Install it and run then run it. Expand the Install DoD Certificates pane and click on the Certificate tab:



Next select the three DoD Root CA certs from the list of certificates and click “PEM” under Export tool group:



After clicking the “PEM” export button choose a location to export the certificates to and click OK. This should have created three .cer files in the directory you selected. Open up a command prompt and navigate to that directory.

Here we will use the keytool command to import the certificates into a truststore. Run the following

commands to import the three certificates:

> keytool -importcert -file DoD_Root_CA_2__0x05__DoD_Root_CA_2.cer -alias DODRoot2 -keystore truststore.jks -storepass changeit
>
> keytool -importcert -file DoD_Root_CA_3__0x01__DoD_Root_CA_3.cer -alias DODRoot3 -keystore truststore.jks -storepass changeit
>
> keytool -importcert -file DoD_Root_CA_4__0x01__DoD_Root_CA_4.cer -alias DODRoot4 -keystore truststore.jks -storepass changeit

This will create a truststore.jks file with a password of 'changeit' in the current working directory. It will contain the three DoD Root Certs, you can see this by running:

> keytool -list -keystore truststore.jks

Which should list out something like:

> Your keystore contains 3 entries
>
> dodroot4, Sep 23, 2016, trustedCertEntry, Certificate fingerprint (SHA1): B8:26:9F:25:DB:D9:37:EC:AF:D4:C3:5A:98:38:57:17:23:F2:D0:26 dodroot3, Sep 23, 2016, trustedCertEntry, Certificate fingerprint (SHA1): D7:3C:A9:11:02:A2:20:4A:36:45:9E:D3:22:13:B4:67:D7:CE:97:FB dodroot2, Sep 23, 2016, trustedCertEntry, Certificate fingerprint (SHA1): 8C:94:1B:34:EA:1E:A6:ED:9A:E2:BC:54:CF:68:72:52:B4:C9:B5:61

Configure Tomcat to use the Keystore and Truststore

We now have the keystore and truststore files we need, next is to configure tomcat to use them. To do this we must change the /conf/server.xml file. Open the file in add a connector definition like the following:

```
<Connector
    clientAuth="true"
    keystoreFile="path/to/keystore.jks"
    keystorepass="changeit"
    keystoreType="jks"
    truststoreFile="path/to/truststore.jks"
    truststoreType="jks"
    truststorepass="changeit"
    maxThreads="150"
    port="8443"
    protocol="org.apache.coyote.http11.Http11NioProtocol"
    scheme="https"
    secure="true"
    sslProtocol="TLS"
    SSLEnabled="true"
    />
```

You can go here for further definition of all of the attributes: http://tomcat.apache.org/tomcat-7.0-doc/config/http.html

Once this is all done start up tomcat. From a computer that has a CAC reader with a CAC inserted browse to the https://:8443/ url and if everything is configured properly you should be prompted to pick a certificate from the CAC card.

Read CAC enabling Tomcat for Development Purposes online:
https://riptutorial.com/tomcat/topic/6995/cac-enabling-tomcat-for-development-purposes

Chapter 3: Configuring a JDBC Datasource

Introduction

In order to utilize a JDBC datasource, we must first set up a JNDI reference in Tomcat. After the JNDI reference is made, JDBC datasources can be used within our Tomcat server and applications using shared or independent references (Great for `dev`/`staging`/`prod` setup, or removing connection strings/credentials from committed code).

Remarks

Utilizing JNDI and JDBC also affords you to use ORMs like Hibernate or platforms like JPA to define "persistence units" for object and table mapp

Examples

Configuring a server-wide JNDI reference

Inside of your `{CATALINA_HOME}/conf/` folder exists a `server.xml` and `context.xml` file. Each one of these contains similar code, but references different parts of Tomcat to complete the same task.

`server.xml` is server-wide configuration. This is where you can set up HTTPS, HTTP2, JNDI Resources, etc.

`context.xml` is specific to each context in Tomcat, taken from Tomcat's documentation it explains this well:

> The Context element represents a web application, which is run within a particular virtual host. Each web application is based on a Web Application Archive (WAR) file, or a corresponding directory containing the corresponding unpacked contents, as described in the Servlet Specification (version 2.2 or later). For more information about web application archives, you can download the Servlet Specification, and review the Tomcat Application Developer's Guide.

Essentially, it's application-specific configuration.

In order to operate correctly, we'll need to set up a `Resource` in `server.xml` and a reference to that resource inside of `context.xml`.

Inside of `server.xml`'s `<GlobalNamingResources>` element, we'll append a new `<Resource>` which will be our JNDI reference:

```
<GlobalNamingResources>
   <!--
     JNDI Connection Pool for AS400
```

```
      Since it uses an older version of JDBC, we have to specify a validationQuery
      to bypass errornous calls to isValid() (which doesn't exist in older JDBC)
    -->
    <Resource name="jdbc/SomeDataSource"
              auth="Container"
              type="javax.sql.DataSource"
              maxTotal="100"
              maxIdle="30"
              maxWaitMillis="10000"
              username="[databaseusername]"
              password="[databasepassword]"
              driverClassName="com.ibm.as400.access.AS400JDBCDriver"
              validationQuery="Select 1 from LIBRARY.TABLE"
              url="jdbc:as400://[yourserver]:[port]"/>
```

In this example, we're using a rather particular datasource (an IBMi - running DB2), which requires a `validationQuery` element set since it's using an older version of JDBC. This example is given as there is very little examples out there, as well as a display of the interoperability that a JDBC system affords you, even for an antiquated DB2 system (as above). Similar configuration would be the same for other popular database systems:

```
<Resource name="jdbc/SomeDataSource"
    auth="Container"
    type="javax.sql.DataSource"
    username="[DatabaseUsername]"
    password="[DatabasePassword]"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql:/[yourserver]:[port]/[yourapplication]"
    maxActive="15"
    maxIdle="3"/>
```

Inside of `context.xml` we'll need to configure a "pointer" towards our jdbc datasource (which we made with a JNDI reference):

```
<Context>
    ...
    <ResourceLink name="jdbc/SomeDataSource"
      global="jdbc/SomeDataSource"
      type="javax.sql.DataSource" />
</Context>
```

Utilizing `ResourceLink` inside of `context.xml` allows us to reference the same datasource across applications and have it configured at the server level for multiple-database systems. (Although it also works just as well with one database)

Using a JNDI reference as a JDBC Resource in Context

```
public void test() {
    Connection conn = null;
    Statement stmt = null;
    try {
        Context ctx = (Context) new InitialContext().lookup("java:comp/env");
        conn = ((DataSource) ctx.lookup("jdbc/SomeDataSource")).getConnection();
```

```
        stmt = conn.createStatement();
        //SQL data fetch using the connection
        ResultSet rs = stmt.executeQuery("SELECT * FROM TABLE");
        while (rs.next()) {
            System.out.println(rs.getString("Id"));
        }
        conn.close();
        conn = null;
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally {

        if (stmt != null || conn != null) try {
            assert stmt != null;
            stmt.close();
        } catch (SQLException ex) {
            // ignore -- as we can't do anything about it here
            ex.printStackTrace();
        }
    }
}
```

Read Configuring a JDBC Datasource online: https://riptutorial.com/tomcat/topic/8911/configuring-a-jdbc-datasource

Chapter 4: Configuring a JNDI datasource

Parameters

Attribute	Details
auth	(String) Specify whether the web Application code signs on to the corresponding resource manager programmatically, or whether the Container will sign on to the resource manager on behalf of the application. The value of this attribute must be Application or Container. This attribute is required if the web application will use a resource-ref element in the web application deployment descriptor, but is optional if the application uses a resource-env-ref instead.
driverClassName	(String) The fully qualified Java class name of the JDBC driver to be used. The driver has to be accessible from the same classloader as the database connection pool jar.
factory	(String) Full class path to the connection datasource factory.
initialSize	(int)The initial number of connections that are created when the pool is started. Default value is 10
maxIdle	(int) The minimum number of established connections that should be kept in the pool at all times. The connection pool can shrink below this number if validation queries fail. Default value is derived from initialSize of 10
maxTotal / maxActive	(int) The maximum number of active connections that can be allocated from this pool at the same time. The default value is 100. Note that this attribute name differs between pool implementations and documentation is often incorrect.
maxWaitMillis / maxWait	(int) The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. Default value is 30000 (30 seconds). Note that this attribute name differs between pool implementations and documentation is often incorrect.
name	(String) Name used to bind to JNDI context.
password	(String) DB connection password.
url	(String) (String) JDBC connection URL.
username	(String) DB connection username.

Attribute	Details
testOnBorrow	(boolean) The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. NOTE - for a true value to have any effect, the validationQuery or validatorClassName parameter must be set to a non-null string. In order to have a more efficient validation, see validationInterval. Default value is false.
validationQuery	(String) The SQL query that will be used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. Example values are SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server)

Remarks

Attributes

The list of available attributes is extensive and fully covered in Tomcat's JDBC Connection Pool reference documentation. Only the attributes used in the examples above are covered in the parameters section here.

DBCP vs Tomcat JDBC Connection Pool

Many locations in reference documentation refer to use of DBCP connection pools. The history on which connection pool implementation is actually being used in Tomcat, by default, is complex and confusing. It depends on specific version of Tomcat being used. It's best to specify the factory explicitly.

Reference Documentation

- Tomcat 8 JDNI Resources HOW-TO - JDBC Data Sources
- Tomcat 8 JNDI Datasource HOW-TO - Examples
- Tomcat 8 JDBC Connection Pool Reference
- Tomcat 8 Context Resource Links Reference

Examples

JNDI Datasource for PostgreSQL & MySQL

Declare JNDI resource in tomcat's server.xml, using the Tomcat JDBC connection pool:

```
<GlobalNamingResources>
    <Resource name="jdbc/DatabaseName"
```

```
              factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
              auth="Container"
              type="javax.sql.DataSource"
              username="dbUser"
              password="dbPassword"
              url="jdbc:postgresql://host/dbname"
              driverClassName="org.postgresql.Driver"
              initialSize="20"
              maxWaitMillis="15000"
              maxTotal="75"
              maxIdle="20"
              maxAge="7200000"
              testOnBorrow="true"
              validationQuery="select 1"
              />
</GlobalNamingResources>
```

And reference the JNDI resource from Tomcat's web context.xml:

```
  <ResourceLink name="jdbc/DatabaseName"
   global="jdbc/DatabaseName"
   type="javax.sql.DataSource"/>
```

If using MySQL, change URL, driver, and validation query:

```
  url="jdbc:mysql://host:3306/dbname"
  driverClassName="com.mysql.jdbc.Driver"
  validationQuery="/* ping */ SELECT 1"
```

JNDI Encrypted credentials

In the JNDI declaration you may want to encrypt the username and password.

You have to implement a custom datasource factory in order to be able to decrypt the credentials.

In `server.xml` replace `factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"` by `factory="cypher.MyCustomDataSourceFactory"`

Then define your custom factory :

```
package cypher;

import java.util.Enumeration;
import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.Name;
import javax.naming.RefAddr;
import javax.naming.Reference;
import javax.naming.StringRefAddr;

import org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory;

public class MyCustomDataSourceFactory extends BasicDataSourceFactory {
    //This must be the same key used while encrypting the data
```

```
    private static final String ENC_KEY = "aad54a5d4a5dad2ad1a2";

    public MyCustomDataSourceFactory() {}

    @Override
    public Object getObjectInstance(final Object obj, final Name name, final Context nameCtx,
final Hashtable environment) throws Exception {
        if (obj instanceof Reference) {
            setUsername((Reference) obj);
            setPassword((Reference) obj);
        }
        return super.getObjectInstance(obj, name, nameCtx, environment);
    }

    private void setUsername(final Reference ref) throws Exception {
        findDecryptAndReplace("username", ref);
    }

    private void setPassword(final Reference ref) throws Exception {
        findDecryptAndReplace("password", ref);
    }

    private void findDecryptAndReplace(final String refType, final Reference ref) throws
Exception {
        final int idx = find(refType, ref);
        final String decrypted = decrypt(idx, ref);
        replace(idx, refType, decrypted, ref);
    }

    private void replace(final int idx, final String refType, final String newValue, final
Reference ref) throws Exception {
        ref.remove(idx);
        ref.add(idx, new StringRefAddr(refType, newValue));
    }

    private String decrypt(final int idx, final Reference ref) throws Exception {
        return new CipherEncrypter(ENC_KEY).decrypt(ref.get(idx).getContent().toString());
    }

    private int find(final String addrType, final Reference ref) throws Exception {
        final Enumeration enu = ref.getAll();
        for (int i = 0; enu.hasMoreElements(); i++) {
            final RefAddr addr = (RefAddr) enu.nextElement();
            if (addr.getType().compareTo(addrType) == 0) {
                return i;
            }
        }

        throw new Exception("The \"" + addrType + "\" name/value pair was not found" + " in
the Reference object. The reference Object is" + " "
                + ref.toString());
    }
}
```

Of course you need an utility to encrypt the username and password ;

```
    package cypher;

import java.io.UnsupportedEncodingException;
import java.security.spec.AlgorithmParameterSpec;
```

```
import java.security.spec.KeySpec;

import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;

public class CipherEncrypter {

    Cipher ecipher;

    Cipher dcipher;

    byte[] salt = {
            (byte) 0xA9, (byte) 0x9B, (byte) 0xC8, (byte) 0x32, (byte) 0x56, (byte) 0x35,
(byte) 0xE3, (byte) 0x03
    };

    int iterationCount = 19;

    /**
     * A java.security.InvalidKeyException with the message "Illegal key size or default
parameters" means that the cryptography strength is limited; the unlimited strength
juridiction policy files are not in the correct location. In a JDK,
     * they should be placed under ${jdk}/jre/lib/security
     *
     * @param passPhrase
     */
    public CipherEncrypter(final String passPhrase) {
        try {
            // Create the key
            SecretKeyFactory factory = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
            KeySpec spec = new PBEKeySpec(passPhrase.toCharArray(), salt, 65536, 256);
            SecretKey tmp = factory.generateSecret(spec);
            //            SecretKey secret = new SecretKeySpec(tmp.getEncoded(), "AES");

            // Create the ciphers
            ecipher = Cipher.getInstance(tmp.getAlgorithm());
            dcipher = Cipher.getInstance(tmp.getAlgorithm());

            final AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt,
iterationCount);

            ecipher.init(Cipher.ENCRYPT_MODE, tmp, paramSpec);
            dcipher.init(Cipher.DECRYPT_MODE, tmp, paramSpec);

        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String encrypt(final String str) {
        try {
            final byte[] utf8 = str.getBytes("UTF8");
            byte[] ciphertext = ecipher.doFinal(utf8);
            return new sun.misc.BASE64Encoder().encode(ciphertext);
        }
        catch (final javax.crypto.BadPaddingException e) {
```

```
            //
        }
        catch (final IllegalBlockSizeException e) {
            //
        }
        catch (final UnsupportedEncodingException e) {
            //
        }
        catch (Exception e) {
            //
        }

        return null;
    }

    public String decrypt(final String str) {
        try {

            final byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);
            return new String(dcipher.doFinal(dec), "UTF-8");
        }
        catch (final javax.crypto.BadPaddingException e) {
            //TODO
        }
        catch (final IllegalBlockSizeException e) {
            //TODO
        }
        catch (final UnsupportedEncodingException e) {
            //TODO
        }
        catch (final java.io.IOException e) {
            //TODO
        }
        return null;
    }

    public static void main(final String[] args) {

        if (args.length != 1) {
            System.out.println("Error : you have to pass exactly one argument.");
            System.exit(0);
        }
        try {
            //This key is used while decrypting.
            final CipherEncrypter encrypter = new CipherEncrypter("aad54a5d4a5dad2ad1a2");
            final String encrypted = encrypter.encrypt(args[0]);
            System.out.println("Encrypted :" + encrypted);

            final String decrypted = encrypter.decrypt(encrypted);
            System.out.println("decrypted :" + decrypted);
        }
        catch (final Exception e) {

            e.printStackTrace();
        }

    }
}
```

When you have encrypted values for username and password, replace the clear ones in

`server.xml`.

Note that the encrypter should be in an obfuscated jar to keep the private key hidden (or you can also pass the key as an argument of the programm).

Read Configuring a JNDI datasource online: https://riptutorial.com/tomcat/topic/4652/configuring-a-jndi-datasource

Chapter 5: Embedding into an application

Examples

Embed tomcat using maven

```
      <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.1</version>
            <executions>
                <execution>
                    <id>tomcat-run</id>
                      <goals>
                        <goal>exec-war-only</goal>
                       </goals>
<!--This phase is for creating jar file.You can customize configuration -->
                        <phase>package</phase>
                    <configuration>
                        <path>/WebAppName</path>
                        <enableNaming>false</enableNaming>
                        <finalName>WebAppName.jar</finalName>
                    </configuration>
                </execution>
            </executions>
<!--This configuration is for running application in your ide-->
            <configuration>
                <port>8020</port>
                <path>/webappName</path>
        <!--These properties are optional-->
                <systemProperties>
                    <CATALINA_OPTS>-Djava.awt.headless=true -Dfile.encoding=UTF-8
                        -server -Xms1536m -Xmx1536m
                        -XX:NewSize=256m -XX:MaxNewSize=256m -XX:PermSize=256m
                        -XX:MaxPermSize=512m -XX:+DisableExplicitGC
                        -XX:+UseConcMarkSweepGC
                        -XX:+CMSIncrementalMode
                        -XX:+CMSIncrementalPacing
                        -XX:CMSIncrementalDutyCycleMin=0
                        -XX:-TraceClassUnloading
                    </CATALINA_OPTS>
                </systemProperties>
            </configuration>
        </plugin>
```

You can run the above tomcat in your ide using goal `tomcat:run`. If you run `package` goal it will create a jar file in your target folder which can create tomcat instance itself and run.

Using `</CATALINA_OPTS>` you can specify properties like permgen max and min size, Garbage Collection mechanism etc.which are completely optional.

Read Embedding into an application online: https://riptutorial.com/tomcat/topic/3876/embedding-into-an-application

Chapter 6: Https configuration

Examples

SSL/TLS Configuration

HTTPS

HTTPS (also called HTTP over TLS,[1][2] HTTP over SSL,[3] and HTTP Secure[4][5]) is a protocol for secure communication over a computer network which is widely used on the Internet. HTTPS consists of communication over Hypertext Transfer Protocol (HTTP) within a connection encrypted by Transport Layer Security or its predecessor, Secure Sockets Layer. The main motivation for HTTPS is authentication of the visited website and protection of the privacy and integrity of the exchanged data.

SSL

Image result for what is ssl SSL (Secure Sockets Layer) is the standard security technology for establishing an encrypted link between a web server and a browser. This link ensures that all data passed between the web server and browsers remain private and integral. SSL is a security protocol. Protocols describe how algorithms should be used.

TLS

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), both of which are frequently referred to as 'SSL', are cryptographic protocols designed to provide communications security over a computer network.

SSL Certificate

All browsers have the capability to interact with secured web servers using the SSL protocol. However, the browser and the server need what is called an SSL Certificate to be able to establish a secure connection.

SSL Certificates have a key pair: a public and a private key. These keys work together to establish an encrypted connection. The certificate also contains what is called the “subject,” which is the identity of the certificate/website owner.

How Does the SSL Certificate Create a Secure Connection

1. When a browser attempts to access a website that is secured by SSL, the browser and the web server establish an SSL connection using a process called an “SSL Handshake”

2. Essentially, three keys are used to set up the SSL connection: the public, private, and session keys.

Steps to Establish a Secure Connection

1. Browser connects to a web server (website) secured with SSL (https). Browser requests that the server identify itself.

2. Server sends a copy of its SSL Certificate, including the server’s public key.

3. Browser checks the certificate root against a list of trusted CAs and that the certificate is unexpired, unrevoked, and that its common name is valid for the website that it is connecting to. If the browser trusts the certificate, it creates, encrypts, and sends back a symmetric session key using the server’s public key.

4. Server decrypts the symmetric session key using its private key and sends back an acknowledgement encrypted with the session key to start the encrypted session.

5. Server and Browser now encrypt all transmitted data with the session key.

SSL/TLS and Tomcat

It is important to note that configuring Tomcat to take advantage of secure sockets is usually only necessary when running it as a stand-alone web server.

And if running Tomcat primarily as a Servlet/JSP container behind another web server, such as Apache or Microsoft IIS, it is usually necessary to configure the primary web server to handle the SSL connections from users.

Certificates

In order to implement SSL, a web server must have an associated Certificate for each external interface (IP address) that accepts secure connections.Certificate as a "digital driver's license".

1. This "driver's license" is cryptographically signed by its owner, and is therefore extremely difficult for anyone else to forge

2. Certificate is typically purchased from a well-known Certificate Authority (CA) such as VeriSign or Thawte

In many cases, however, authentication is not really a concern. An administrator may simply want to ensure that the data being transmitted and received by the server is private and cannot be snooped by anyone who may be eavesdropping on the connection. Fortunately, Java provides a relatively simple command-line tool, called keytool, which can easily create a "self-signed" Certificate. Self-signed Certificates are simply user generated Certificates which have not been officially registered with any well-known CA, and are therefore not really guaranteed to be authentic at all

Prepare the Certificate Keystore

Tomcat currently operates only on JKS, PKCS11 or PKCS12 format keystores.

JKS:

The JKS format is Java's standard "Java KeyStore" format, and is the format created by the

keytool command-line utility. This tool is included in the JDK

PKCS11/ PKCS12

The PKCS12 format is an internet standard, and can be manipulated via (among other things) OpenSSL and Microsoft's Key-Manager.

To create a new JKS keystore from scratch, containing a single self-signed Certificate, execute the following from a terminal command line:

```
$ keytool -genkey -alias tomcat -keyalg RSA
```

This command will create a new file, in the home directory of the user under which you run it, named ".keystore".

To specify a different location or filename, add the -keystore parameter, followed by the complete pathname to your keystore file as .

```
$ Keytool -genkey -alias tomcat -keyalg RSA  -keystore \path\to\my\dir\<keystore-file-name>
```

After executing this command, you will first be prompted for

1. keystore password
2. and for general information about this Certificate, such as company, contact name, and so on.

Finally, you will be prompted for the key password, which is the password specifically for this Certificate (as opposed to any other Certificates stored in the same keystore file).

If everything was successful, you now have a keystore file with a Certificate that can be used by your server.

Edit the Tomcat Configuration File

Tomcat can use two different implementations of SSL:

1. JSSE implementation provided as part of the Java runtime (since 1.4)

   The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication

   The JSSE API was designed to allow other SSL/TLS protocol and Public Key Infrastructure (PKI) implementations to be plugged in seamlessly. Developers can also provide alternative logic to determine if remote hosts should be trusted or what authentication key material should be sent to a remote host.

2. APR implementation, which uses the OpenSSL engine by default.

The exact configuration details of Connector depend on which implementation is being used.

```
<!-- Default in configuration file ..-->

<Connector protocol="HTTP/1.1"  port="8080" .../>
```

To define a Java (JSSE) connector, regardless of whether the APR library is loaded or not, use one of the following:

```
<!-- Define a HTTP/1.1 Connector on port 8443, JSSE NIO implementation -->

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" .../>
```

Alternatively, to specify an APR connector (the APR library must be available) use:

```
<!-- Define a HTTP/1.1 Connector on port 8443, APR implementation -->
<Connector protocol="org.apache.coyote.http11.Http11AprProtocol"
           port="8443" .../>
```

to configure the Connector in the $CATALINA_BASE/conf/server.xml file

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector
           protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"/>
```

If you change the port number here, you should also change the value specified for the redirectPort attribute on the non-SSL connector. This allows Tomcat to automatically redirect users who attempt to access a page with a security constraint specifying that SSL is required, as required by the Servlet Specification.

Configure in web.xml for particular project

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SUCTR</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Installing a Certificate from a Certificate Authority

1. Create a local Certificate Signing Request (CSR)
2. Create a local self-signed Certificate as described above

3. The CSR is then created with

```
$ keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr
   -keystore <your_keystore_filename>
```

Now you have a file called certreq.csr that you can submit to the Certificate Authority

Importing the Certificate

Now that you have your Certificate and you can import it into you local keystore. First of all you have to import a Chain Certificate or Root Certificate into your keystore. After that you can proceed with importing your Certificate.

1. Download a Chain Certificate from the Certificate Authority you obtained the Certificate from

2. Import the Chain Certificate into your keystore

```
$ keytool -import -alias root -keystore <your_keystore_filename>
   -trustcacerts -file <filename_of_the_chain_certificate>
```

3. And finally import your new Certificate

```
keytool -import -alias tomcat -keystore <your_keystore_filename>
   -file <your_certificate_filename>
```

Reference : Here

Read Https configuration online: https://riptutorial.com/tomcat/topic/6026/https-configuration

Chapter 7: Tomcat Virtual Hosts

Remarks

Host Manager is a web application inside of Tomcat that creates/removes *Virtual Hosts* within Tomcat.

A *Virtual Host* allows you to define multiple hostnames on a single server, so you can use the same server to handles requests to, for example, `ren.myserver.com` and `stimpy.myserver.com`.

Unfortunately documentation on the GUI side of the Host Manager doesn't appear to exist, but documentation on configuring the virtual hosts manually in `context.xml` is here:

http://tomcat.apache.org/tomcat-7.0-doc/virtual-hosting-howto.html.

The full explanation of the `Host` parameters you can find here:

http://tomcat.apache.org/tomcat-7.0-doc/config/host.html.

Adapted from my answer: http://stackoverflow.com/a/26248511/6340

Examples

Tomcat Host Manager Web Application

Tomcat's Host Manager application by default is located at http://localhost:8080/host-manager, but is not accessible until a user is given permission in the `conf/tomcat-users.xml` file. The file needs to have:

1. A `manager-gui` role
2. A user with this role

For example:

```
<tomcat-users>
    ...
    <role rolename="manager-gui"/>
    ....
    <user username="host-admin" password="secretPassword" roles="manager-gui"/>
</tomcat-users>
```

Adding a Virtual Host via the Tomcat Host Manager Web Application

Once you have access to the host-manager, the GUI will let you add a virtual host.

> **Note:** In Tomcat 7 and 8, adding a virtual host via the GUI **does not write the vhost to config files**. You will need to manually edit the `server.xml` file to have the vhost available after a restart. See http://tomcat.apache.org/tomcat-7.0-doc/virtual-hosting-

howto.html for further info on the `<Host>` tag in `server.xml`

At a minimum you need the `Name` and `App Base` fields defined. Tomcat will then create the following directories:

```
{CATALINA_HOME}\conf\Catalina\{Name}
{CATALINA_HOME}\{App Base}
```

- `App Base` will be where web applications will be deployed to the virtual host. Can be relative or absolute.
- `Name` is usually the fully-qualified domain name (e.g. `ren.myserver.com`)
- `Alias` can be used to extend the `Name` also where two addresses should resolve to the same host (e.g. `www.ren.myserver.com`). Note that this needs to be reflected in DNS records.

The checkboxes are as follows:

- `Auto Deploy`: Automatically redeploy applications placed into App Base. Dangerous for Production environments!
- `Deploy On Startup`: Automatically boot up applications under App Base when Tomcat starts
- `Deploy XML`: Determines whether to parse the application's `/META-INF/context.xml`
- `Unpack WARs`: Unpack WAR files placed or uploaded to the App Base, as opposed to running them directly from the WAR.
- **Tomcat 8** `Copy XML`: Copy an application's `META-INF/context.xml` to the App Base/XML Base on deployment, and use that exclusively, regardless of whether the application is updated. Irrelevant if `Deploy XML` is false.
- `Manager App`: Add the manager application to the Virtual Host (Useful for controlling the applications you might have underneath `ren.myserver.com`)

Adapted from my answer: http://stackoverflow.com/a/26248511/6340

Adding a Virtual Host to server.xml

Once a virtual host has been added via the web application, directories will exist at:

```
{CATALINA_HOME}\conf\Catalina\{Name}
{CATALINA_HOME}\{App Base}
```

To persist the virtual host after a restart, the `server.xml` file must be updated with the configuration. A `Host` element needs to be added inside the `Engine` element, similar to this:

```
<Engine name="Catalina" ...>
  ...
  <Host name="my-virtual-app" appBase="virtualApp" autoDeploy="true" unpackWARs="true" ... />
</Engine>
```

The attributes in the `Host` element should reflect the selections made in the host manager GUI (see the Host documentation for details), but can be changed. Note that the `Manager App` option in the GUI does not correspond to any `Host` attribute.

Read Tomcat Virtual Hosts online: https://riptutorial.com/tomcat/topic/6235/tomcat-virtual-hosts

Chapter 8: Tomcat(x) Directories Structures

Examples

Directory Structure in Ubuntu (Linux)

After installing Tomcat with apt-get on Ubuntu xx.xx, Tomcat creates and uses these directories:

$cd /etc/tomcat6/

```
├── Catalina
│   └── localhost
│       ├── ROOT.xml
│       └── solr.xml -> ../../../solr/solr-tomcat.xml
├── catalina.properties
├── context.xml
├── logging.properties
├── policy.d
│   ├── 01system.policy
│   ├── 02debian.policy
│   ├── 03catalina.policy
│   ├── 04webapps.policy
│   ├── 05solr.policy -> /etc/solr/tomcat.policy
│   └── 50local.policy
├── server.xml
├── tomcat-users.xml
└── web.xml
```

$cd /usr/share/tomcat6

```
├── bin
│   ├── bootstrap.jar
│   ├── catalina.sh
│   ├── catalina-tasks.xml
│   ├── digest.sh
│   ├── setclasspath.sh
│   ├── shutdown.sh
│   ├── startup.sh
│   ├── tomcat-juli.jar -> ../../java/tomcat-juli.jar
│   ├── tool-wrapper.sh
│   └── version.sh
├── defaults.md5sum
├── defaults.template
└── lib
    ├── annotations-api.jar -> ../../java/annotations-api-6.0.35.jar
    ├── catalina-ant.jar -> ../../java/catalina-ant-6.0.35.jar
    ├── catalina-ha.jar -> ../../java/catalina-ha-6.0.35.jar
    ├── catalina.jar -> ../../java/catalina-6.0.35.jar
    ├── catalina-tribes.jar -> ../../java/catalina-tribes-6.0.35.jar
    ├── commons-dbcp.jar -> ../../java/commons-dbcp.jar
    ├── commons-pool.jar -> ../../java/commons-pool.jar
    ├── el-api.jar -> ../../java/el-api-2.1.jar
    ├── jasper-el.jar -> ../../java/jasper-el-6.0.35.jar
    ├── jasper.jar -> ../../java/jasper-6.0.35.jar
    ├── jasper-jdt.jar -> ../../java/ecj.jar
```

```
    ├── jsp-api.jar -> ../../java/jsp-api-2.1.jar
    ├── servlet-api.jar -> ../../java/servlet-api-2.5.jar
    ├── tomcat-coyote.jar -> ../../java/tomcat-coyote-6.0.35.jar
    ├── tomcat-i18n-es.jar -> ../../java/tomcat-i18n-es-6.0.35.jar
    ├── tomcat-i18n-fr.jar -> ../../java/tomcat-i18n-fr-6.0.35.jar
    └── tomcat-i18n-ja.jar -> ../../java/tomcat-i18n-ja-6.0.35.jar
```

$cd /usr/share/tomcat6-root/

```
└── default_root
    ├── index.html
    └── META-INF
        └── context.xml
```

$cd /usr/share/doc/tomcat6

```
├── changelog.Debian.gz -> ../libtomcat6-java/changelog.Debian.gz
├── copyright
└── README.Debian.gz -> ../tomcat6-common/README.Debian.gz
```

$cd /var/cache/tomcat6

```
├── Catalina
│   └── localhost
│       ├── _
│       └── solr
│           └── org
│               └── apache
│                   └── jsp
│                       ├── admin
│                       │   ├── form_jsp.class
│                       │   ├── form_jsp.java
│                       │   ├── get_002dproperties_jsp.class
│                       │   ├── get_002dproperties_jsp.java
│                       │   ├── index_jsp.class
│                       │   ├── index_jsp.java
│                       │   ├── schema_jsp.class
│                       │   ├── schema_jsp.java
│                       │   ├── stats_jsp.class
│                       │   ├── stats_jsp.java
│                       │   ├── threaddump_jsp.class
│                       │   └── threaddump_jsp.java
│                       ├── index_jsp.class
│                       └── index_jsp.java
└── catalina.policy
```

$cd /var/lib/tomcat6

```
├── common
│   └── classes
├── conf -> /etc/tomcat6
├── logs -> ../../log/tomcat6
├── server
│   └── classes
├── shared
│   └── classes
```

```
├── webapps
│   └── ROOT
│       ├── index.html
│       └── META-INF
│           └── context.xml
└── work -> ../../cache/tomcat6
```

$cd /var/log/tomcat6

```
├── catalina.2013-06-28.log
├── catalina.2013-06-30.log
├── catalina.out
├── catalina.out.1.gz
└── localhost.2013-06-28.log
```

$cd /etc/default

```
├── tomcat7
```

Read Tomcat(x) Directories Structures online: https://riptutorial.com/tomcat/topic/5964/tomcat-x--directories-structures

Credits

S. No	Chapters	Contributors
1	Getting started with tomcat	CodeWarrior, Community, Stefan
2	CAC enabling Tomcat for Development Purposes	David Harris
3	Configuring a JDBC Datasource	Shawn S.
4	Configuring a JNDI datasource	alain.janinm, kaliatech
5	Embedding into an application	udaybhaskar
6	Https configuration	Girish Kumar
7	Tomcat Virtual Hosts	brasskazoo
8	Tomcat(x) Directories Structures	Girish Kumar