

# Stateless Session Bean

Stateless beans are used in the case when the process or action can be completed in one go. In this case, object state will not be preserved in a series of linked invocation. Each invocation of bean will be unique in itself in a Stateless Session Bean. A stateless session bean is represented by **@Stateless** annotation.

Here is an example of Simple Stateless Session Bean:

## StatelessBean.java

```
@Stateless
public class StatelessBean implements StatelessBeanRemote
{
    public void SimpleFunction()
    {
        System.out.println("This is a stateless session bean");
    }
}
```

As can be seen in the code, @Stateless defines a simple POJO as a Stateless Session bean.

The above gives a very high level idea of what is a stateless session bean and how is it implemented using EJB3 annotations. While working with examples we will try to make you understand stateless session beans in-depth.

## Stateless Session Bean Sample

This sample application will take you through the basics of Stateless Session Bean. This application will demonstrate how annotations like @Stateless, @Resource, @PostConstruct, @PreDestroy are used in an EJB3 application.

The application walks through a authentication page, where the user has to authenticate to move to the resource page of Apache Geronimo. In case of a new user, the user has to go through the registration process. Later the user will be directed to login page again once the registration is done. In the login page the bean class will check for the username and password entered by the user against a database.

Basically a Stateless Session EJB is used whenever there is a single step process and maintaining a session is obsolete. In this sample the user registration form is a one step process and hence we have used stateless session bean for its implementation. The login page is a misnomer and should not be considered as an implementation for stateless session EJB. This is because once logged in you have to maintain the session and stateless session beans are not meant to maintain the session.

To run this tutorial, as a minimum you will be required to have installed the following prerequisite software.

- Sun JDK 5.0+ (J2SE 1.5)
- Eclipse 3.3.1.1 (Eclipse Classic package of Europa distribution), which is platform specific
- Web Tools Platform (WTP) 2.0.1
- Data Tools Platform (DTP) 1.5.1
- Eclipse Modeling Framework (EMF) 2.3.1
- Graphical Editing Framework (GEF) 3.3.1

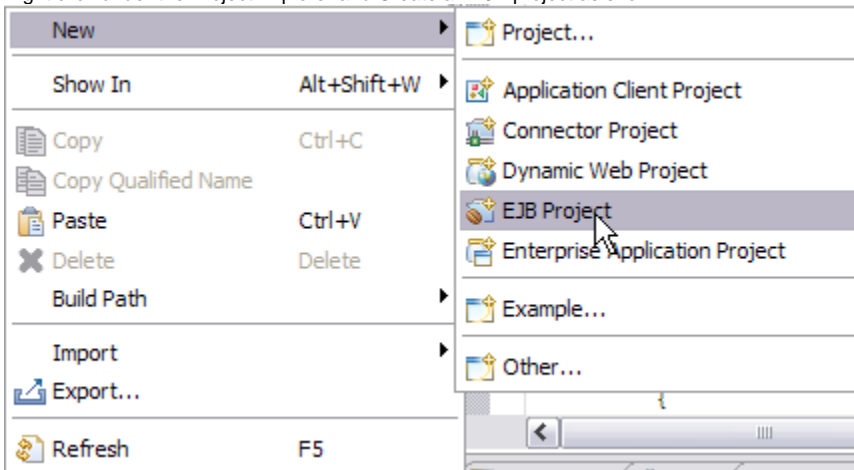
Details on installing eclipse are provided in the [Development environment](#) section.

This tutorial is organized in the following sections:

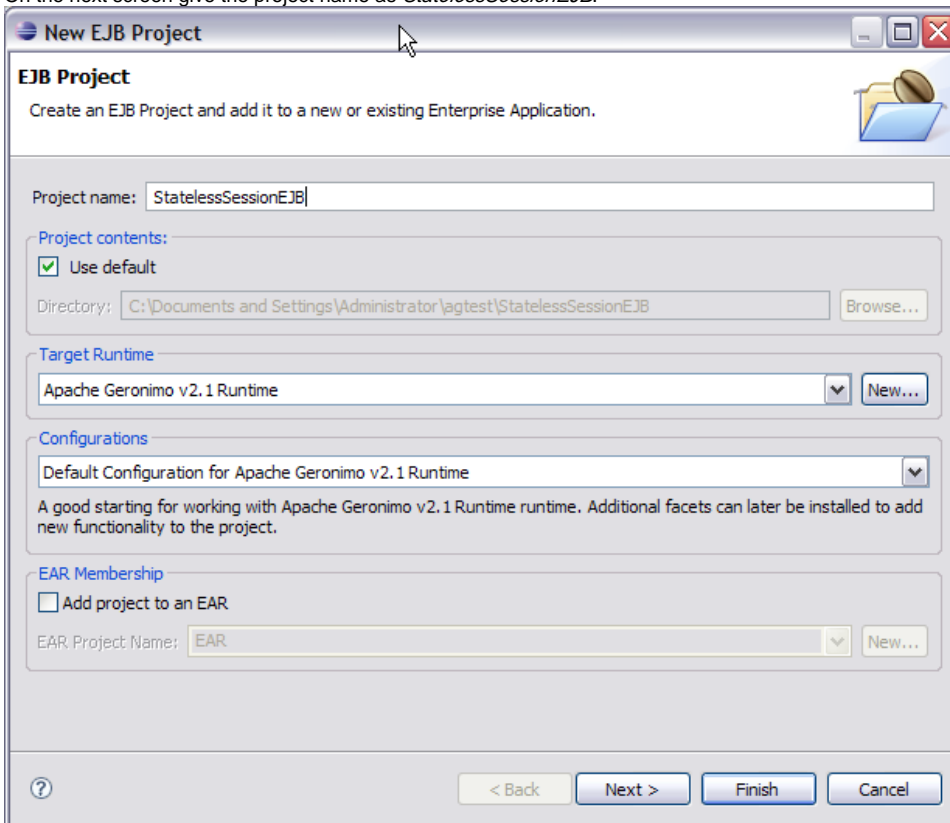
- [Creating a Stateless Session EJB project](#)
- [Creating a database using the administrative console](#)
- [Creating a datasource using Administrative Console](#)
- [Creating application client](#)
- [Few more configurations](#)
- [Deploy and Run](#)

## Creating a Stateless Session EJB project

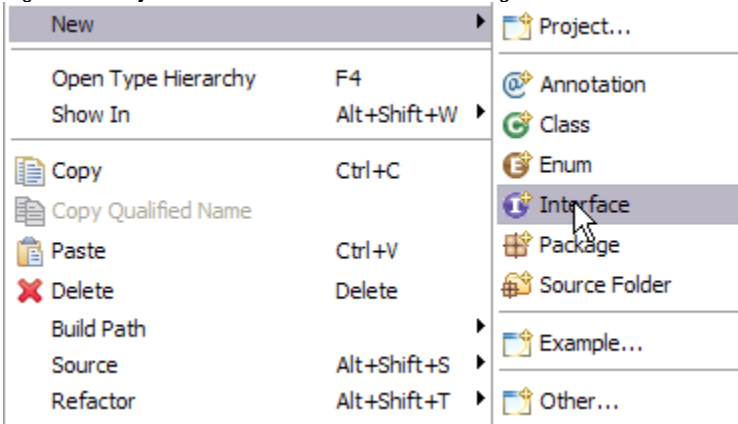
1. Right click under the Project Explorer and Create an EJB project as shown.



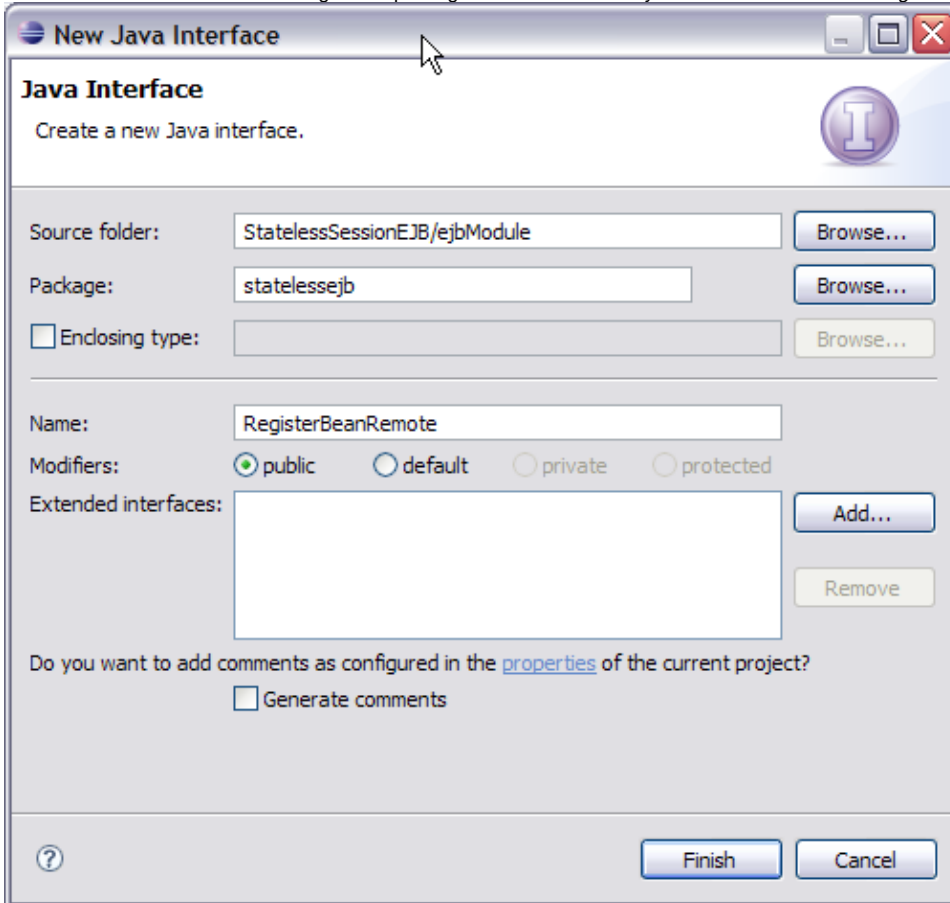
2. On the next screen give the project name as *StatelessSessionEJB*.



3. Right click on **ejbModule** and create a new Interface *RegisterBeanRemote*.



4. On the New Java Interface window give the package name as *statelessejb* and Interface name as *RegisterBeanRemote*.



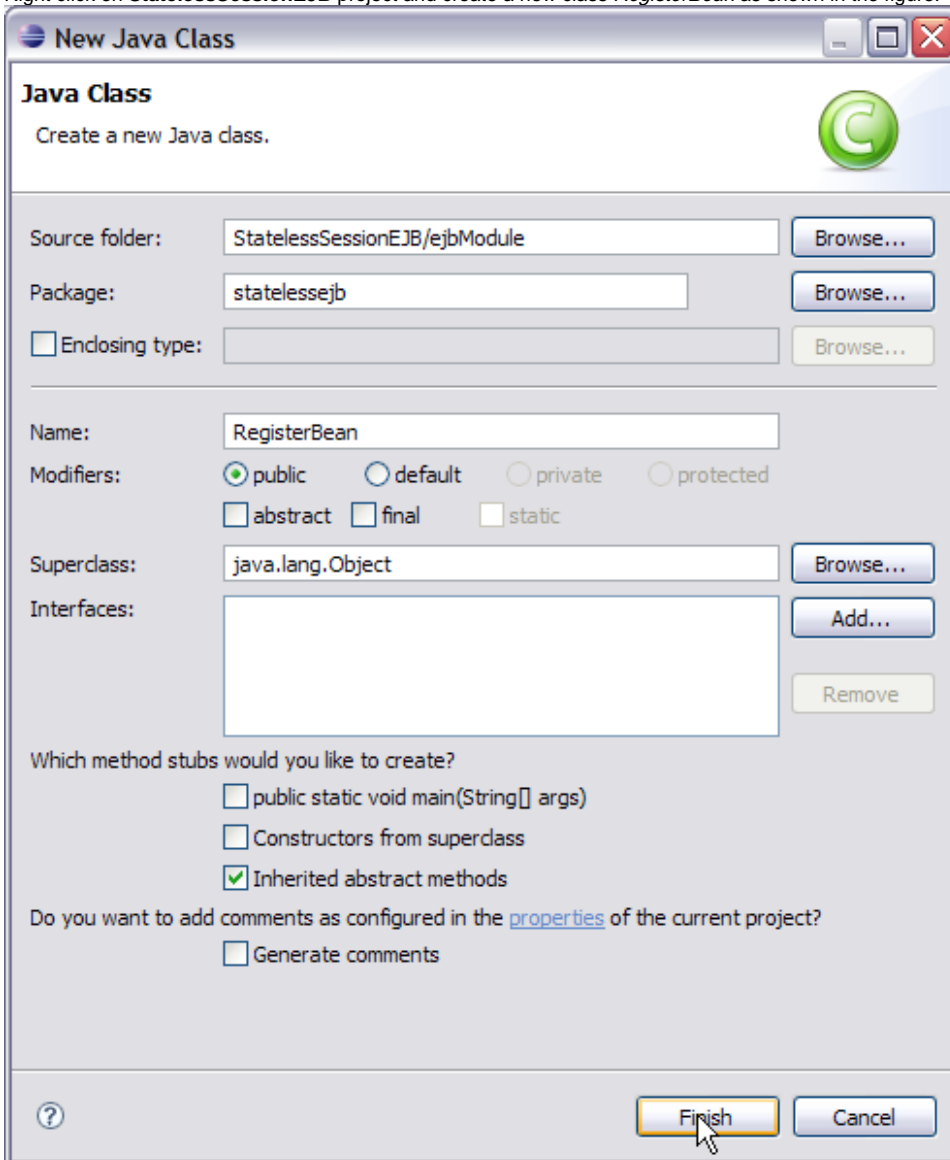
5. Populate the *RegisterBeanRemote* interface with the following code.

#### RegisterBeanRemote.java

```
package statelessejb;
import javax.ejb.Remote;
@Remote
public interface RegisterBeanRemote
{
    public void register(String FirstName, String LastName, String Sex, String UserName, String Password);
    public boolean verify(String username, String password);
}
```

In the above code @Remote is a metadata annotation which marks the interface to be a Remote Interface. Metadata annotations are used to declare a class, interface or functions to have particular attributes. In this case the interface is marked to be a Remote Interface.

6. Right click on **StatelessSessionEJB** project and create a new class *RegisterBean* as shown in the figure.



7. Populate the class *RegisterBean* with the following data:

#### RegisterBean.java

```
package statelessejb;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.sql.DataSource;
@Stateless
public class RegisterBean implements RegisterBeanRemote{
    @Resource(name="jdbc/userds")
    private DataSource datasource;
    private Connection dbconnect;
```

```

@PostConstruct
public void initialize()
{
    try{
        dbconnect= dbsource.getConnection();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

public void register(String FirstName, String LastName, String Sex, String UserName, String
Password)
{
    try
    {
        String insert="INSERT INTO USERINFO (" + "FIRSTNAME," + "LASTNAME," + "SEX," +
"USERNAME," + "PASSWORD)" + " VALUES(?,?,?,?,?)";
        PreparedStatement ps=dbconnect.prepareStatement(insert);
        ps.setString(1,FirstName);
        ps.setString(2,LastName);
        ps.setString(3,Sex);
        ps.setString(4,UserName);
        ps.setString(5>Password);
        int rs =ps.executeUpdate();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

public boolean verify(String username, String password)
{
    boolean ret=false;
    String select="SELECT * FROM USERINFO";
    try{
        PreparedStatement ps=dbconnect.prepareStatement(select);
        ResultSet rs= ps.executeQuery();
        while(rs.next())
        {
            String str=(rs.getString("USERNAME")).trim();
            String str1=(rs.getString("PASSWORD")).trim();
            if (str.compareTo(username)==0)
            {
                if(str1.compareTo(password)==0)
                {
                    {
                        ret=true;
                        break;
                    }
                }
            }
            else
                ret=false;
        }
        else
            ret=false;
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    System.out.println(ret);
    return ret;
}

@PreDestroy
public void destroy(){
    try
    {
        dbconnect.close();
        dbconnect=null;
    }
}

```

```

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

```

Lets try to understand this code.

- The very first line has an annotation **@Stateless** which declares the POJO to be a Stateless Session Bean. This annotation tells the container that the class is a Stateless Session Bean class. The container now automatically takes care of the various services like pooling, thread safety, transactions and so on.
- The second line suggests that the bean class implements the remote interface. Every EJB application has to have at least one interface. In this case we are having a remote interface.
- Next we have the **@Resource(name="jdbc/userds")** annotation which suggests *jdbc/userds* datasource being injected on to EJB. This is called dependency injection. The main idea behind dependency injection is that a component should call the resource through interfaces. This in turn will create a loosely coupled architecture between component and resource. Dependency injection is actually JNDI lookup in reverse order. In JNDI lookup the Bean looks up the resources itself and that is why it has to be hardcoded in the bean code itself whereas in Dependency Injection the Container reads the Bean and finds out what resources are required by the bean class. Later it injects the resources at runtime. How to create *jdbc/userds* datasource is discussed in the next section.
- Next interesting part is the **@PostConstruct** annotation. This annotation is used for lifecycle callbacks. A lifecycle callback method is used by container to inform the bean class of the state transitions in the bean lifecycle. **@PostConstruct** annotation is used once a bean instance is created. In our example we have our bean instance created and Dependency injected. Later **@PostConstruct** callback is invoked and a connection to the datasource is established.
- Next we have the **register** function which is used to populate the database **USERINFO** with user information. This function uses **PreparedStatement** to query the database.
- Next is the **verify** function which is used for the authentication of user credentials.
- @PreDestroy** is again a lifecycle callback which suggests to release the resources before the bean is destroyed.

Warning

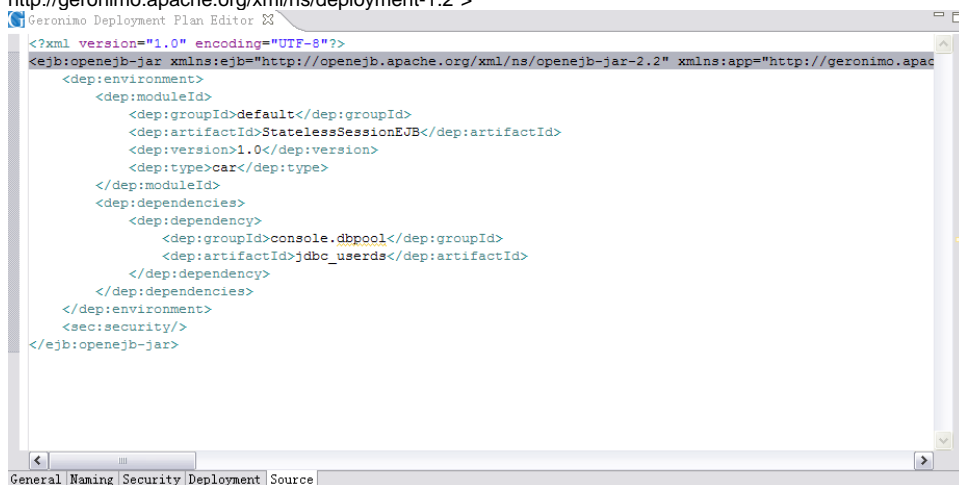


Due to some limitations in Geronimo Eclipse Plugin the generated deployment plan(*openejb-jar.xml*) does not have the correct namespace. Replace the existing namespace as shown in the figure with the following

```

<openejb-jar xmlns="http://www.openejb.org/xml/ns/openejb-jar-2.2" xmlns:nam="http://geronimo.apache.org/xml/ns/naming-1.2" xmlns:
pkggen="http://www.openejb.org/xml/ns/pkggen-2.0" xmlns:sec="http://geronimo.apache.org/xml/ns/security-1.2" xmlns:sys="
http://geronimo.apache.org/xml/ns/deployment-1.2">

```

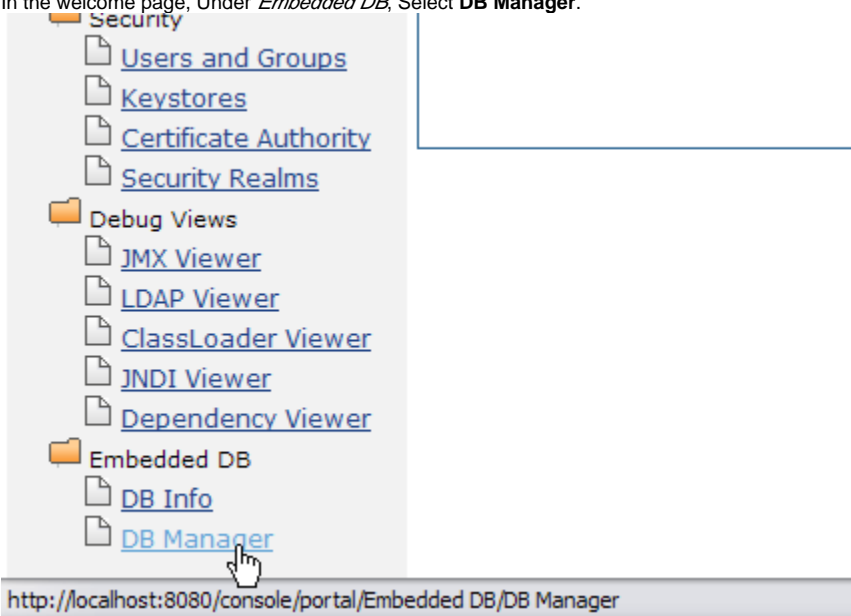


This completes the development of EJB project.

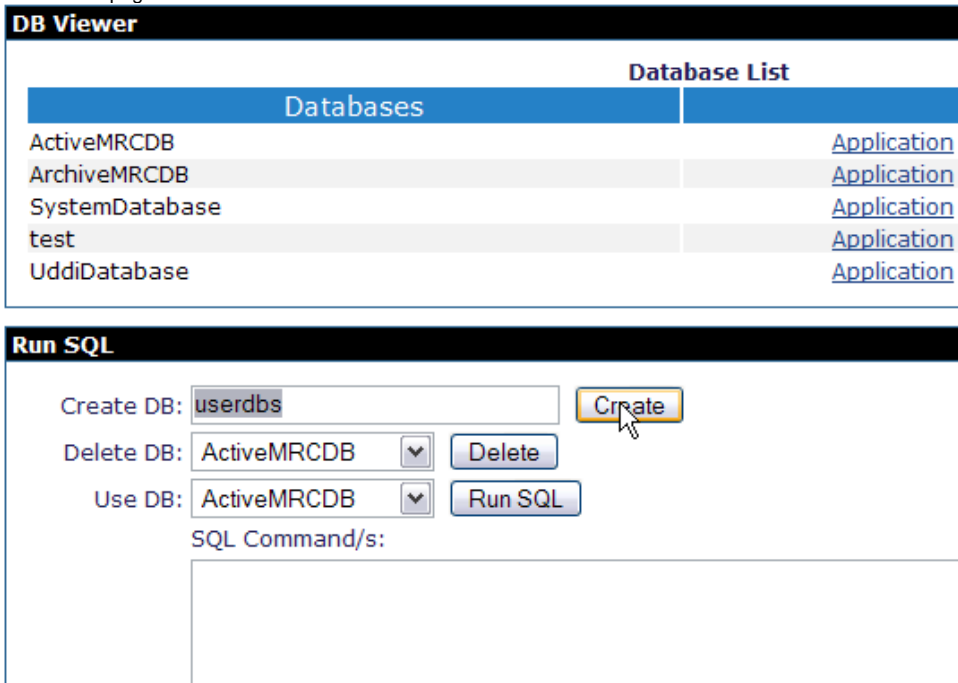
## Creating a database using the administrative console

1. Start the server and launch the Administrative Console using the URL <http://localhost:8080/console>.
2. Enter default username and password.

3. In the welcome page, Under *Embedded DB*, Select **DB Manager**.



4. On the next page create a database *userdbs* and Select **Create**.



5. Once done you can see the **userdbs** database listed in DB Viewer portlet under Databases. This confirms that the database has been successfully created.

DB Viewer		
Database List		
Databases		View
ActiveMRCDB		<a href="#">Application</a>
ArchiveMRCDB		<a href="#">Application</a>
SystemDatabase		<a href="#">Application</a>
test		<a href="#">Application</a>
UddiDatabase		<a href="#">Application</a>
userdbs		<a href="#">Application</a>

6. As shown in the figure under *Use DB*, select **userdbs** from the dropdown box.

**Run SQL**

Create DB:

Delete DB: ActiveMRCDB

Use DB: ActiveMRCDB

ActiveMRCDB  
 ArchiveMRCDB  
 SystemDatabase  
 test  
 UddiDatabase  
**userdbs**

7. Run the query as shown in the figure. This query will create table **USERINFO** with the columns *FIRSTNAME*, *LASTNAME*, *SEX*, *USERNAME*, *PASSWORD*.

**Run SQL**

Create DB:

Delete DB: ActiveMRCDB

Use DB: userdbs

SQL Command/s:

```

Create table userinfo(
  firstname char(20),
  lastname char(20),
  sex char(20),
  username char(20),
  password char(20)
)

```

#### CreateTable.sql

```

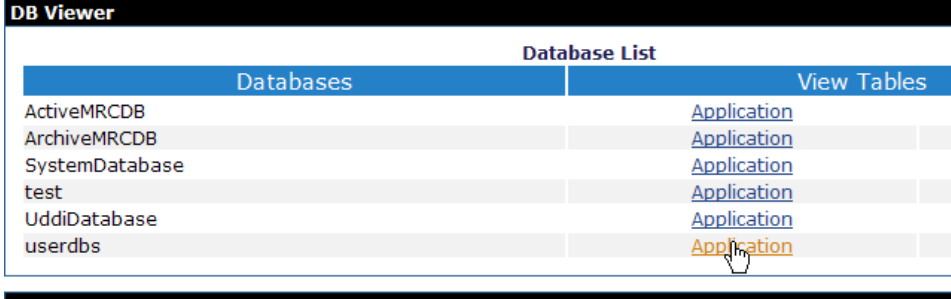
CREATE TABLE USERINFO
(
  FIRSTNAME char(20),
  LASTNAME char(20),
  SEX char(7),
  USERNAME char(20),

```



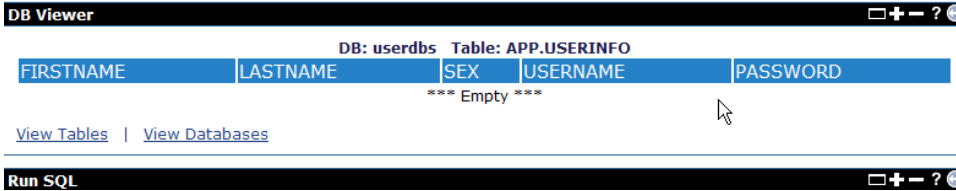
```
PASSWORD char(20)
)
```

8. To verify the table creation is successful. Under *View Tables* for **userdbs** Database, Select **Application**.



Database List	
Databases	View Tables
ActiveMRCDB	<a href="#">Application</a>
ArchiveMRCDB	<a href="#">Application</a>
SystemDatabase	<a href="#">Application</a>
test	<a href="#">Application</a>
UddiDatabase	<a href="#">Application</a>
userdbs	<a href="#">Application</a>

9. The next window will show the table **USERINFO** associated with **userdbs** Database.



DB: userdbs Table: APP.USERINFO				
FIRSTNAME	LASTNAME	SEX	USERNAME	PASSWORD
*** Empty ***				

[View Tables](#) | [View Databases](#)

## Creating a datasource using Administrative Console

1. Start the server and Launch the Administrative Console using the URL <http://localhost:8080/console>.
2. Enter default username and password.
3. Once in the welcome page. In console navigation, Under *Services*, Select **Database Pools**.



**APACHE GERONIMO**  
Server Console

**Console Navigation** | **Welcome**

- Welcome
- Server
  - [Information](#)
  - [Java System Info](#)
  - [Server Logs](#)
  - [Shutdown](#)
  - [Web Server](#)
  - [Thread Pools](#)
  - [Apache HTTP](#)
  - [JMS Server](#)
  - [Monitoring](#)
- Services
  - [Repository](#)
  - [Database Pools](#)
  - [JMS Resources](#)

**Welcome to the Apache Geronimo™ Administration Console!**

The administration console provides a convenient, user friendly way to administer many aspects of the Geronimo Server. It is currently a work in progress, and will continue to evolve over time. The navigation panel on the left-hand side of the screen provides easy access to the individual tasks available in the console.

This space is the main content area where the real work happens. Each view contains one or more portlets (self contained view fragments) that typically include a link for help in the header. Look at the top of this portlet for an example and try it out.

The references on the right are provided so that you can learn more about Apache Geronimo, its capabilities, and what might be coming in future releases.

Mailing lists are available to get involved in the development of Apache Geronimo or to ask questions of the community:

4. On the next screen, Create a new database pool using Geronimo database pool wizard.

### Database Pools

This page lists all the available database pools.

For each pool listed, you can click the **usage** link to see examples of how to use

Name	Deployed As
MonitoringClientDS	Server-wide
NoTxDatasource	Server-wide
SystemDatasource	Server-wide
jdbc/ActiveDS	Server-wide
jdbc/ArchiveDS	Server-wide
jdbc/testds	Server-wide
jdbc/juddiDB	org.apache.geronimo.configs/uddi-tomcat/2.1/car

Create a new database pool:

- ◆ [Using the Geronimo database pool wizard](#)
- ◆ [Import from JBoss 4](#)
- ◆ [Import from WebLogic 8.1](#)

5. On the next screen give the name as suggested in the figure. This will initiate the process to create a Derby Embedded XA datasource.

### Database Pools

Create Database Pool -- Step 1: Select Name and Database

**Name of Database Pool:**    
A name that is different than the name for any other database (the name please).

**Database Type:**    
The type of database the pool will connect to.

6. Select the Driver jar and give the database name as *userds* (Remember this is the database we created in the previous step). The rest of fields can be set to default.

This page edits a new or existing database pool.

**Pool Name:** jdbc/userds

A name that is different than the name for any other database pools in the server (no spaces in the name please).

**Pool Type:** TranQL Embedded XA Resource Adapter for Apache Derby

A resource adaptor that provides access to an embedded Apache Derby database with XA support.

#### Basic Connection Properties

**Driver JAR:**

The JAR(s) required to make a connection to the database. Use CTRL-click or SHIFT-click to select multiple jars.

The JAR(s) should already be installed under GERONIMO/repository/ (or [Download a Driver](#))

**Database Name:** userdbs

Name of the database to connect to.

**Password:**

#### 7. Select **Deploy** to deploy the connector plan.

**Create Database:** true

This config-property is currently ignored by Derby.

Flag indicating that the database should be created if it does not exist. This is a

#### Connection Pool Parameters

**Pool Min Size:** 0

The minimum number of connections in the pool. The default is 0.

**Pool Max Size:** 10

The maximum number of connections in the pool. The default is 10.

**Blocking Timeout:** (in milliseconds)

The length of time a caller will wait for a connection. The default is 5000.

**Idle Timeout:** (in minutes)

How long a connection can be idle before being closed. The default is 15.

[Cancel](#)

- Once done you can see the Database Pool **jdbc/userds** listed in the available database pools.

**Database Pools**

This page lists all the available database pools.

For each pool listed, you can click the **usage** link to see examples of how to use the

Name	Deployed As
MonitoringClientDS	Server-wide
NoTxDatasource	Server-wide
SystemDatasource	Server-wide
jdbc/ActiveDS	Server-wide
jdbc/ArchiveDS	Server-wide
jdbc/testds	Server-wide
<b>jdbc/userds</b>	Server-wide
jdbc/juddiDB	org.apache.geronimo.configs/uddi-tomcat/2.1/car

Create a new database pool:

- ◆ [Using the Geronimo database pool wizard](#)
- ◆ [Import from JBoss 4](#)
- ◆ [Import from WebLogic 8.1](#)

## Creating application client

- Create a new dynamic Web Project with the name *ApplicationClient*.
- Right click on **WebContent** and create the following `login.jsp`:

```
login.jsp

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome to Apache Geronimo</title>
</head>
<body bgcolor="white">
<form method="post" action="passCredentials.jsp">
<h2 align="center"><font color="blue">Welcome to Apache Geronimo</font></h2>
<h2 align="center"><font color="blue">Enter your credentials</font></h2>
Enter your Username
<input type="text" name="username" size=20><br>
Enter your Password
<input type="password" name="password" size=20><br>
<input type="submit" value="Login">
<a href="http://localhost:8080/ApplicationClient/register.jsp">NewUser</a>
</form>
</body>
</html>
```

This form is the login page for our application. Once the user enters his/her credentials, these are passed to another jsp `passCredentials.jsp` (checkout the action in the form tag) to verify the authenticity of user credentials. In case the user is new he has to go through the registration process. This statement `<a href="http://localhost:8080/ApplicationClient/register.jsp">NewUser</a>` is used to route the user to registration page.

```
passCredentials.jsp

<%@ page import="java.util.Properties, javax.naming.Context, javax.naming.InitialContext, statelessejb.
```

```

RegisterBeanRemote" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body bgcolor="white">
<%!boolean i; %>
<%
Properties prop=new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.openejb.client.RemoteInitialContextFactory");
prop.put("java.naming.provider.url", "ejbd://localhost:4201");
Context context = new InitialContext(prop);
RegisterBeanRemote myejb=(RegisterBeanRemote)context.lookup("RegisterBeanRemote");
String s= request.getParameter("username");
String s1= request.getParameter("password");
i=myejb.verify(s,s1);
%>
<%
    if (i==true){
%>
<jsp:forward page="/resources.jsp"></jsp:forward>
<%
    } else {
%>
    <jsp:forward page="/login.jsp"></jsp:forward>
<%
    }
%>
</body>
</html>

```

**<%!boolean i; %>** is a declaration for a global variable. The other part is a scriptlet which does a JNDI lookup to the remote interface. Later the user credentials are passed to **verify** function. In case the credentials are correct user is routed to the resources page else he is redirected to login page to re-login.

Why is the lookup name RegisterBeanRemote??



This will be explained in the deploy an run section

#### resources.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome to Apache Geronimo</title>
</head>
<body bgcolor="white">
<h3><font color="blue">Welcome to Apache Geronimo Resource Center</font></h3>
Apache Geronimo Home Page
<a href="http://geronimo.apache.org">Apache Geronimo Home page</a><br>
Join our mailing list
<a href="http://geronimo.apache.org/mailling-lists.html">Apache Geronimo Mailing List</a><br>
Come and Contribute to Apache Geronimo V2.1 Documentation
<a href="http://cwiki.apache.org/GMOxDOC22/">Apache Geronimo V2.1 Documentation</a>
</body>
</html>

```

This is the final page of the application which displays the various resources.

#### register.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome to Apache Geronimo</title>
</head>
<body bgcolor="white">
<script type="text/javascript">
function valForm()
{
    if (this.document.form1.username.value==" " || this.document.form1.username.value=="")
    {
        alert("You cannot leave the field blank");
        this.document.form1.firstname.focus();
        return false;
    }
    else
    {
        return true;
    }
}
</script>
<h2 align="center"><font color="blue">Welcome to Apache Geronimo User Registration</font></h2>
<form method="post" name="form1" action="passVariables.jsp" onSubmit=" return valForm();">
FirstName
<INPUT type="text" name="firstname" SIZE=20><br>
LastName
<INPUT type="text" name="lastname" SIZE=20 ><br>
Sex<br>
<input type="radio" name="sex" value="male"> Male
<br>
<input type="radio" name="sex" value="female"> Female
<br>
Select a UserName<br>
<input type="text" name="username" size=20><br>
Select a Password<br>
<input type="password" name="password" size=20><br>
<br>
<input type="submit" value="Submit">
</form>
</body>
</html>

```

This page is the registration page. Once the user fills up the form and presses Submit a **valform()** function is called which validates the field username and password. In case any of them is empty an alert is sent which suggests empty fields in the form. If all is fine `passVariables.jsp` is called.

#### passVariables.jsp

```

<%@ page import="java.util.Properties, javax.naming.Context, javax.naming.InitialContext, statelessejb.
RegisterBeanRemote" %>

<html>
<head>

<meta http-equiv="Refresh" content="5;URL=http://localhost:8080/ApplicationClient/login.jsp">
<title>Welcome to Apache Geronimo</title>
</head>

<%
Properties prop=new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.openejb.client.RemoteInitialContextFactory");
prop.put("java.naming.provider.url", "ejbd://localhost:4201");
Context context = new InitialContext(prop);
RegisterBeanRemote myejb=(RegisterBeanRemote)context.lookup("RegisterBeanRemote");
String s= request.getParameter("firstname");
String s1= request.getParameter("lastname");
String s2= request.getParameter("sex");

```

```
String s3= request.getParameter("username");
String s4= request.getParameter("password");
myejb.register(s,s1,s2,s3,s4);
%>

<h3 align="center"><font color="blue">Thank you for registering with Apache Geronimo</font></h3>
<h3 align="center"><font color="blue">Redirecting to Login Page</font></h3>
</html>
```

In this page the fields are retrieved from `register.jsp` and the `register` function in the bean class is called to populate the database. The code `<meta http-equiv="Refresh" content="5;URL=http://localhost:8080/ApplicationClient/login.jsp">` suggests to wait for 5 seconds and then move to `login.jsp`

#### index.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<jsp:forward page="/login.jsp" />
</body>
</html>
```

## Few more configurations

1. In the EJB Project. Under *META-INF*, edit `openejb-jar.xml` and add the following

#### datasource dependency

```
<sys:dependencies>
  <sys:dependency>
    <sys:groupId>console.dbpool</sys:groupId>
    <sys:artifactId>jdbc_userds</sys:artifactId>
  </sys:dependency>
</sys:dependencies>
```

Finally the `openejb-jar.xml` will look like this

#### openejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<openejb-jar xmlns="http://www.openejb.org/xml/ns/openejb-jar-2.2"
  xmlns:name="http://geronimo.apache.org/xml/ns/naming-1.2"
  xmlns:pkgen="http://www.openejb.org/xml/ns/pkgen-2.0"
  xmlns:sec="http://geronimo.apache.org/xml/ns/security-1.2"
  xmlns:sys="http://geronimo.apache.org/xml/ns/deployment-1.2">
  <sys:environment>
    <sys:moduleId>
      <sys:groupId>default</sys:groupId>
      <sys:artifactId>StatelessSessionEJB</sys:artifactId>
      <sys:version>1.0</sys:version>
      <sys:type>car</sys:type>
    </sys:moduleId>
    <sys:dependencies>
      <sys:dependency>
        <sys:groupId>console.dbpool</sys:groupId>
        <sys:artifactId>jdbc_userds</sys:artifactId>
      </sys:dependency>
    </sys:dependencies>
  </sys:environment>
```

```
<enterprise-beans/>
</openejb-jar>
```

Where did the above dependencies come from??



To make the datasource visible to EJB we need to add a dependency to the EJB deployment plan that is openejb-jar.xml. The above element can be obtained automatically from Geronimo Database Pool wizard. Select usage against the database pool jdbc/usersd

2. In the WEB Project. Under WEB-INF, Edit geronimo-web.xml and add the following

#### EJB dependency

```
<sys:dependencies>
  <sys:dependency>
    <sys:groupId>default</sys:groupId>
    <sys:artifactId>StatelessSessionEJB</sys:artifactId>
    <sys:version>1.0</sys:version>
    <sys:type>car</sys:type>
  </sys:dependency>
</sys:dependencies>
```

Finally the geronimo-web.xml will look like this

#### geronimo-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web-1.2"
xmlns:nam="http://geronimo.apache.org/xml/ns/naming-1.2"
xmlns:sec="http://geronimo.apache.org/xml/ns/security-1.2"
xmlns:sys="http://geronimo.apache.org/xml/ns/deployment-1.2">
  <sys:environment>
    <sys:moduleId>
      <sys:groupId>default</sys:groupId>
      <sys:artifactId>ApplicationClient</sys:artifactId>
      <sys:version>1.0</sys:version>
      <sys:type>car</sys:type>
    </sys:moduleId>
    <sys:dependencies>
      <sys:dependency>
        <sys:groupId>default</sys:groupId>
        <sys:artifactId>StatelessSessionEJB</sys:artifactId>
        <sys:version>1.0</sys:version>
        <sys:type>car</sys:type>
      </sys:dependency>
    </sys:dependencies>
  </sys:environment>
  <context-root>/ApplicationClient</context-root>
</web-app>
```

3. Right click on the **ApplicationClient** Project and select **properties**. Select Java Build Path->Projects. Click **Add** and add Stateless Session EJB. This is required for the compilation of the Client code.

## Deploy and Run

warning



Due to limitation with Geronimo Eclipse Plugin, you will have to export the Stateless Session EJB project and Web Application project as a jar and war respectively.

1. Export the projects to your local disk as StatelessSessionEJB.jar and ApplicationClient.war.
2. Start the server and launch the Administrative console using <http://localhost:8080/console/>. Enter default username and password.
3. In the welcome page, under Applications. Select Deploy New.



The screenshot shows the Geronimo console interface. On the left is a sidebar with a tree view containing the following items:

- JMS Resources
- Applications
  - Web App WARs
  - System Modules
  - Application EARs
  - EJB JARs
  - J2EE Connectors
  - App Clients
  - Deploy New
  - Plugins
  - Plan Creator
- Security
  - Users and Groups
  - Keystores
  - Certificate Authorities

The main content area contains the following text:

Many resources are available to get involved in the development of Apache Geronimo or to ask questions of the community:

- [user@geronimo.apache.org](mailto:user@geronimo.apache.org) (archives) for general questions related to configuring and using Geronimo
- [dev@geronimo.apache.org](mailto:dev@geronimo.apache.org) (archives) for developers working on Geronimo

So share your experiences with us and let us know how we can make Geronimo even better.

**Thanks for using Geronimo!**

- Browse and Select StatelessSessionEJB.jar. Once done Select Install. This will deploy the EJB application on to server.
- Browse and Select ApplicationClient.war. Once done select Install. This will deploy the Web application on to server.
- Under Applications, Select Web App WARs and Run ApplicationClient project as show in the figure.

## Installed Web Applications

Expert User (enable all actions on Geronimo Provided Components)

Component Name	URL	State
default/ApplicationClient/1.0/car	<a href="#">/ApplicationClient</a>	running
org.apache.geronimo.configs/ca-helper-tomcat/2.1/car	<a href="#">/CAHelper</a>	running
org.apache.geronimo.configs/dojo-legacy-tomcat/2.1/car	<a href="#">/dojo/0.4</a>	running
org.apache.geronimo.configs/dojo-tomcat/2.1/car	<a href="#">/dojo</a>	running

- Once done a welcome page will be launched.

---

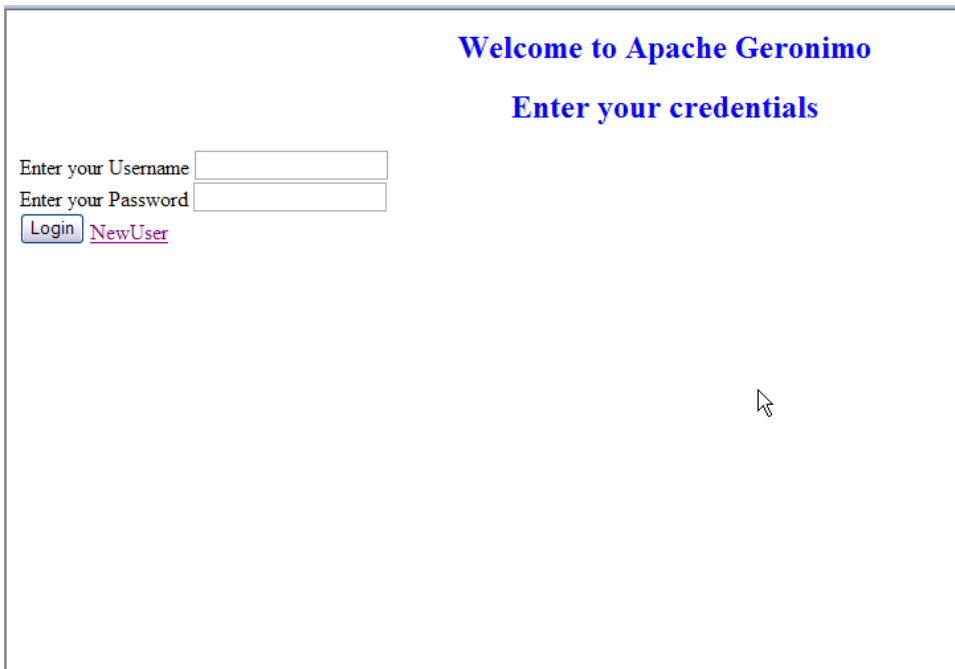
**Welcome to Apache Geronimo**

**Enter your credentials**

Enter your Username

Enter your Password

[NewUser](#)



8. If you are New User. Select **New User** as shown in the figure

---

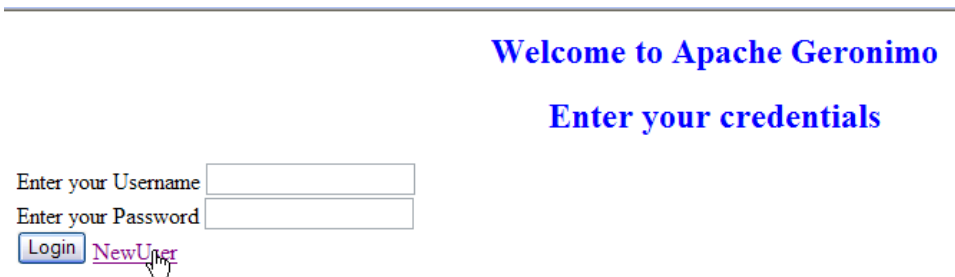
**Welcome to Apache Geronimo**

**Enter your credentials**

Enter your Username

Enter your Password

[NewUser](#)



9. This will display a new registration form. If you leave the mandatory username or password filed empty. You will get the following error message.

---

## Welcome to Apache Geronimo User Registration

FirstName

LastName

Sex

Male

Female

Select a UserName

Select a Password



10. Fill the fields as shown in the figure and Select **Submit**.

---

## Welcome to Apache Geronimo User Registration

FirstName

LastName

Sex

Male

Female

Select a UserName

Select a Password

11. Once done a page will be displayed from where you will be automatically redirected to login page to relogin.

---

**Thank you for registering with Apache Geronimo**

**Redirecting to Login Page**



12. Enter the username and password as chosen by you during registration and select **Login**.

---

**Welcome to Apache Geronimo**

**Enter your credentials**

Enter your Username   
Enter your Password   
 [NewUser](#)

13. The next page is the Apache Geronimo Resource Center.

---

**Welcome to Apache Geronimo Resource Center**

Apache Geronimo Home Page [Apache Geronimo Home page](#)

Join our mailing list [Apache Geronimo Mailing List](#)

Come and Contribute to Apache Geronimo V2.1 Documentation [Apache Geronimo V2.1 Documentation](#)



14. In case you give wrong username and password combination you will be redirected to login page.

---

## Welcome to Apache Geronimo

### Enter your credentials

Enter your Username

Enter your Password

[NewUser](#)