

# The making of MetroSphere, Part 12: Hands-on JSP technology intro: Creating a community weblog

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)

Consultant  
Backstop Media

05 Jun 2003

This tutorial is for developers who want to learn more about using JavaServer Pages (JSP) technology to build applications. It discusses the basics of JSP components in general, their integration with JavaBeans, and the creation and use of custom tag libraries.

## Section 1. Introduction

### Should I take this tutorial?

This tutorial is for developers who want to learn more about using JavaServer Pages (JSP) technology to build applications. It discusses the basics of JSP components in general, their integration with JavaBeans, and the creation and use of custom tag libraries.

The tutorial assumes a basic familiarity with Java syntax, but in-depth Java experience is not required. Familiarity with JDBC, HTML and with XML are helpful, but not required.

### What is this tutorial about?

This series follows the creation of MetroSphere, a community weblog and information marketplace, from project conception to completion. MetroSphere uses IBM WebSphere Portal Server as its base platform, and the portlets that run within the portal typically use JavaServer Pages components to control their output.

JavaServer Pages is a topic that can be confusing if you attempt to get into the details without understanding how the overall technology works. This tutorial is designed to give a reader with no experience in JavaServer Pages technology all of the background necessary to write a fairly sophisticated JSP application without bogging down in details. It chronicles the creation of the basic weblogging, or blogging, application for MetroSphere, detailing the following:

- JSP basics
- Creating the database
- Creating and testing the Blog Entry bean
- Integrating the bean with a JSP page
- Using HTML forms with JSP pages
- Creating and using custom tag libraries

The tutorial is not meant to be an in-depth reference on any of these topics, but will provide you with the big picture and enable you to dive into the details given by other pieces listed in [Resources](#).

## Tools

In order to follow along with this tutorial, you will need to have a JSP-capable environment installed and tested. You can do this in several ways, but the simplest are:

- Install WebSphere Studio. WebSphere Studio provides an incredibly convenient environment for not only building and compiling JSP components and the Java classes that are used with them, but also for testing them in preparation for deployment to a production system. You can download WebSphere Studio from <http://www.ibm.com/developerworks/downloads/ws/wstudio>. Instructions for deploying the completed application to WebSphere Application Server can be found in [Resources](#). The few screenshots in this tutorial show WebSphere Studio Application Developer V4 because that's the version that comes with WebSphere Portal, but the concepts are the same for WebSphere Studio Application Developer Version 5 and for other configurations of WebSphere Studio.

- You will also need a JDBC-compliant database in which to store the weblog entries created by the application. This tutorial uses IBM DB2 Universal Database, which you can download at [https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=db2udbdI&S\\_TA](https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=db2udbdI&S_TA)
- 

## Section 2. Laying the foundation

### Why JavaServer Pages technology?

As you may or may not know, this tutorial is part of a series of pieces describing the development of [MetroSphere](#), a community weblog and information marketplace. The system is built on WebSphere Portal in order to take advantage of the numerous features built into that system.

When I started development on the system, I discovered that WebSphere Portal is based on objects called *portlets*, which are modular applications that are aggregated behind the scenes into a single page. Portlets are Java based, and you can output text directly from them, but it seems that the most efficient way to control the display of information that comes from a portlet is through the use of JavaServer Pages components.

Upon further investigation, it also turned out that all of those great features in the WebSphere Portal user interface were also controlled by JavaServer Pages technology. The flexibility and power of JSP technology was hard to deny. JSPs separate the code behind a page from the presentation of that page, so I could not only make my job easier, I could also enable other people -- such as Dan, our artist -- to manipulate pages without having to worry about the code behind them.

### What you're going to build

The most obvious candidate for the first JSP project was the community weblog system. It was going to be the heart of the site, and it was the first project on my list. Besides, unlike the Portal UI, I could easily work on it in isolation from Portal and concentrate only on the JSPs components. Later, we'll take the application and integrate it into the portlets that will ultimately control it on the site.

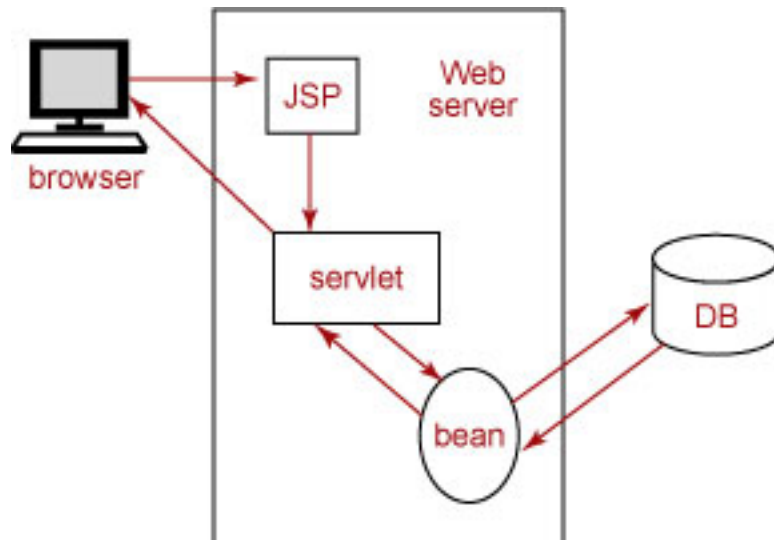
The blogging system we'll build in this tutorial is simple; the user can create an entry using an HTML form and save it to a database. The user can also retrieve a specific entry and make changes to it, and then save it back to the database. Ultimately, we'll

aggregate these entries onto a single page, but while I'll cover everything you need to know in order to do that, I won't actually do it in this tutorial. (You'll have to check out an upcoming installment in the series for that!)

## JSP technology vs. servlets

Before there was JavaServer Pages technology, there were servlets. A servlet is a small Java application that runs on a Web server and outputs its results to the browser. A JavaServer Pages page, on the other hand, sits on a Web server, just as a servlet does, but rather than being a compiled Java application, it's similar to an HTML page in that it's simply text that anyone can edit. The difference is that the first time a JSP page is called by a browser, the Web server compiles that JSP into a servlet, which then returns the result. That servlet sits, ready to go, until the JSP page is changed, and then it's recompiled. In this way, you've got the best of both worlds: presentation isn't tightly integrated with the application, but the JSP doesn't have to be compiled every time it's requested.

The application you're building here actually has several pieces:



The browser makes a request for the JSP component, which actually goes to the compiled version of the JSP, the servlet. The servlet references a Java bean, which has the non-presentation-related functionality. The bean makes a call to the database, say, to retrieve a blog entry, then passes the data back to the servlet, which incorporates it into the page and returns it to the browser.

## The database structure

As long as we're talking about underlying structures, let's take a quick look at the structure of the database. I've created a new database called `BLOG` within `DB2`, and

the following script creates all of the appropriate objects:

```
CREATE SCHEMA BLOGSYSTEM

CREATE TABLE BLOGSYSTEM.ENTRIES (ENTRYID INTEGER NOT NULL, BLOGID INTEGER NOT NULL,
USERID VARCHAR(15) NOT NULL, SUBMITDATE TIMESTAMP NOT NULL, EDITDATE TIMESTAMP,
ENTRYTITLE VARCHAR(255) NOT NULL, SHORTENTRY LONG VARCHAR NOT NULL, COMPLETEENTRY LONG
VARCHAR, ACCESSES INTEGER, FEATURE INTEGER NOT NULL, ACTIVE INTEGER NOT NULL)

ALTER TABLE BLOGSYSTEM.ENTRIES ADD CONSTRAINT BLOGENTRIES_PK PRIMARY KEY (ENTRYID)

CREATE TABLE BLOGSYSTEM.TOPICS (TOPICID INTEGER NOT NULL, TOPICNAME VARCHAR(64) NOT
NULL, TOPICDESC VARCHAR(255))
ALTER TABLE BLOGSYSTEM.TOPICS ADD CONSTRAINT TOPICS_PK PRIMARY KEY (TOPICID)
INSERT INTO BLOGSYSTEM.TOPICS (TOPICID, TOPICNAME, TOPICDESC) VALUES (0, 'No Topic',
'No topic set')
INSERT INTO BLOGSYSTEM.TOPICS (TOPICID, TOPICNAME, TOPICDESC) VALUES (1, 'Java',
'The Java language')
INSERT INTO BLOGSYSTEM.TOPICS (TOPICID, TOPICNAME, TOPICDESC) VALUES (2, 'Databases',
'All database systems')
INSERT INTO BLOGSYSTEM.TOPICS (TOPICID, TOPICNAME, TOPICDESC) VALUES (3, 'Miscellaneous',
'Any other topics')

CREATE TABLE BLOGSYSTEM.TOPIC_ENTRIES (TOPICID INTEGER NOT NULL, ENTRYID INTEGER NOT
NULL)
ALTER TABLE BLOGSYSTEM.TOPIC_ENTRIES ADD CONSTRAINT TOPIC_ENTRIES_PK PRIMARY KEY
(TOPICID, ENTRYID)
ALTER TABLE BLOGSYSTEM.TOPIC_ENTRIES ADD CONSTRAINT TOPC_ENTR_TOPC_FK FOREIGN KEY
(TOPICID) REFERENCES BLOGSYSTEM.TOPICS(TOPICID) ON DELETE NO ACTION ON UPDATE NO ACTION

CREATE TABLE BLOGSYSTEM.DUMMY (PLACEHODLLDER CHAR(1))
INSERT INTO BLOGSYSTEM.DUMMY VALUES ('X')
CREATE SEQUENCE BLOGSYSTEM.ID_SEQ START WITH 1 INCREMENT BY 1 NO MAXVALUE NO CYCLE
CACHE 24
```

Personally, I find the easiest way to run these scripts is to copy and paste them into the DB2 Command Line Interpreter, but if you do that, remember to take out the line feeds within each command; they're only there because of formatting restrictions for the tutorial.

Once the database has been created, it's time to create the project.

## Create the project

If you're using a development tool such as WebSphere Studio, you will need to create a new project and make sure that it has access to the JDBC driver classes.

Start by choosing **File > New > Project > Web > Web Project > Next**.

**Create a Web Project**

**Define the Web Project**

Create a Web project and add it to a new or existing Enterprise Application project.

Project name: JSPProj

Use default location

Location: D:\Program Files\IBM\Application Developer\workspace\JSPProj

Enterprise Application project name: JSPProjEAR

Context root: JSPProj

Create CSS file

< Back   Next >   Finish   Cancel

Enter the name of your project. By default, WebSphere Studio will either want to make the project part of `DefaultEAR` or part of the last EAR you specified, but make sure that you give it its own EAR file to simplify deployment later. Choose **Next**, and then **Next** again. Click the **Libraries** tab to add the JDBC driver classes to the project. Click **Add External Jars**, and, if you're using DB2, find `db2java.zip` -- it's usually in `SQLLIB/java` -- and click **Open**. Then click **Finish**.

If you aren't using WebSphere Studio, make sure that `db2java.zip` (or whatever file contains your DB2 drivers) is on the `CLASSPATH` for both your development environment and for your JSP environment.

---

## Section 3. JSP basics

### The structure of a JavaServer Pages component

This page has two scriptlets, shown in bold:

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

  <% java.lang.String entryTitle;
      entryTitle = "Things to come";

  <h1><% out.print(entryTitle); %></h1>

</body>
</html>
```

The first scriptlet declares a simple `String` and initializes it, and the second simply prints it to the screen. The `out` object, similar to `System.out`, is predefined, so when the user calls this page, the browser actually receives a page that looks like this:

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

  <h1>Things to come</h1>

</body>
</html>
```

Notice that the scriptlets have been replaced by their output (if any) and that the browser simply receives HTML text. This makes it possible to view JSP output on any browser.

### Simplifying output

Outputting text is so common on a JSP page that there is a simple way of doing it. Rather than creating a scriptlet and using `out.print()`, you can simply use the output abbreviation, as shown here:

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

  <% java.lang.String entryTitle;
     entryTitle = "Things to come"; %>

  <h1><%= entryTitle %></h1>

</body>
</html>
```

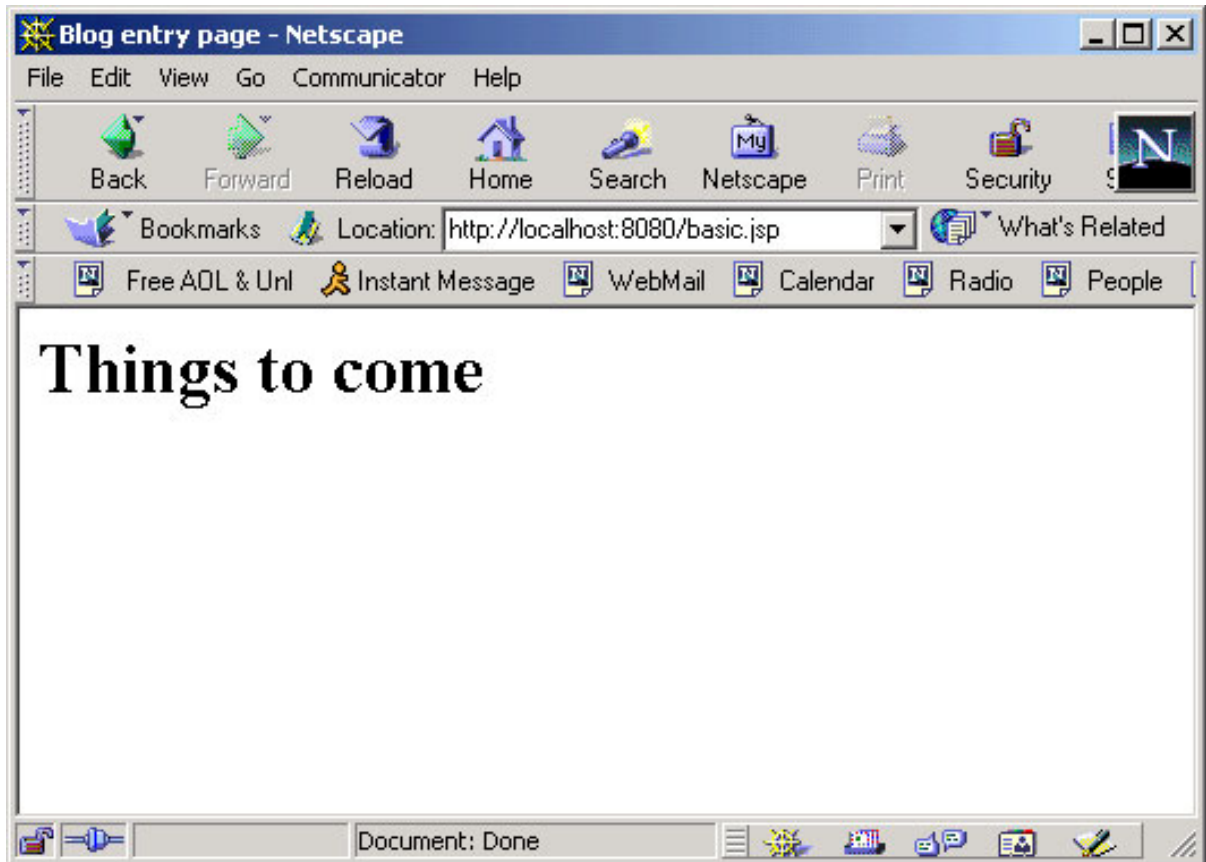
This page is equivalent to the one on the previous page in every way, but uses the `<%= %>` abbreviation for output. Any expression can be output to the page in this way.

Notice that while the scriptlet requires the use of semicolons, as in any Java class, the output abbreviation actually forbids them.

## Testing the page

Before you go any further, test the page and make sure that everything's working properly. In WebSphere Studio, save the file as `basic.jsp` and right-click it in the Navigator pane on the left-hand side. Choose Run on server.

You should see the results of the simple HTML page.



Now let's look at how beans fit in to the equation.

## Creating a simple object bean

Part of the power of JavaServer Pages technology comes from the ability to create and use objects within the page. These objects are normally in the form of *beans*. I'll get more into what Java beans are in [What is a bean?](#), but for now just understand that it's a Java object.

The easiest way to create the new object is to simply declare it using the `jsp:useBean` action:

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

  <jsp:useBean id="entryTitle" class="java.lang.String" />

  <% entryTitle = "Things to come"; %>

  <h1><%= entryTitle %></h1>

</body>
```

```
</html>
```

If you've never used XML before, the `jsp:` prefix might look a bit strange to you. That prefix is what signals to the server that this element is special, an action that needs to be evaluated rather than simply output like the HTML tags. In this case, it simply tells the server to create a new object of the class `java.lang.String`, and call it `entryTitle`. Notice that you're then using the `entryTitle` object just as though you had declared it explicitly, as you did in previous panels.

In this way, you can create all sorts of objects, which you can then reference by the name you give them in the `id` attribute.

## Coding logic into the page

Now, this doesn't look tremendously impressive; after all, it's just a Java statement.

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

  <jsp:useBean id="entryTitle" class="java.lang.String" />
  <jsp:useBean id="isLoggedIn" class="java.lang.String" />

  <% entryTitle = "Things to come";
     isLoggedIn = "true"; %>

  <h1><%= entryTitle %></h1>
  <% if (isLoggedIn.equals("true")) {
     out.print("<p>Please add your entry.</p>");
   } else {
     out.print("<p>Please log in.</p>");
   } %>

</body>
</html>
```

But let's look at this a little more closely. In an if-then statement, if the expression evaluates to true, everything between those first two brackets will be executed. In this case, it's just a simple `print()` statement, but it doesn't have to be. Consider this rewriting of the page, with non-relevant sections snipped out:

```
...

  <% entryTitle = "Things to come";
     isLoggedIn = "true"; %>

  <h1><%= entryTitle %></h1>
  <% if (isLoggedIn.equals("true")) { %>

     <p>Please add your entry.</p>

  <% } else { %>
```

```
        <p>Please log in.</p>
    <% } %>
</body>
</html>
```

Here the server is still going to do whatever's between those brackets, but in this case it's going to output the HTML element asking the user to add his or her entry, so the user should wind up with a page that looks like this:

```
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

    <h1>Things to come</h1>

    <p>Please add your entry.</p>

</body>
</html>
```

In this way, scriptlets can include or exclude entire sections of a page based on logic.

## Tag libraries

Before we move on to the actual application, let's just take a quick peek at one more way to use JSP components to streamline coding.

JavaServer Pages technology enables the creation of custom tag libraries, which let a developer simply add a special tag to the page. When the server encounters the tag, it performs the code specified and outputs the result. We'll look at tag libraries in detail in [Tag libraries](#), but for now let's take a quick look at an example:

```
        <%@ taglib uri="blog.tld" prefix="blog" %>
<html>
<head>
<title>Blog entry page</title>
</head>
<body>

    <jsp:useBean id="entryTitle" class="java.lang.String" />
    <jsp:useBean id="isLoggedIn" class="java.lang.String" />

    <% entryTitle = "Things to come";
        isLoggedIn = "true"; %>
```

```
<h1><%= entryTitle %></h1>
<blog:greeting />

</body>
</html>
```

Now let's move on to building a more useful bean.

---

## Section 4. Adding the entry bean

### What is a bean?

You can probably find entire articles and books just on Java beans, but what it comes down to is this: a Java bean is a class that has specific properties and specific methods for getting and setting those properties. Consider this very simple bean meant to represent a blog entry topic:

```
package com.metrosphere.util;

public class TopicBean {

    public int topicId;
    public int getTopicId(){
        return topicId;
    }
    public void setTopicId(int newTopicId){
        topicId = newTopicId;
    }

    public String topicName;
    public String getTopicName(){
        return topicName;
    }
    public void setTopicName(String newTopicName){
        topicName = newTopicName;
    }
}
```

This bean has two properties, `topicId` and `topicName`. Each has a "getter" method (`getTopicId()` and `getTopicName()`) and a "setter" method (`setTopicId()` and `setTopicName()`), so they are both "read-write" properties. If, for example, you wanted to make the `topicId` property read-only, you could simply eliminate the `setTopicId()` method.

What's special about these methods is that they match the name of the property. Even if I didn't know anything else about the bean, I know that to get the `topicId` property, I use the `getTopicId()` method. If there were a `isSelected` property, I

know without having to know anything else that to retrieve it, I would need the `getIsSelected()` method.

This uniformity in naming property access methods is what enables tools to work together without having to provide enormous amounts of additional information, and without having to know how the properties are actually stored. The `topicId` property could just as easily have been called `TOPICID` or `_TOPICIDENT` or `_BOB`. As long as the method name is correct, the bean will interoperate.

Of course, this bean is almost ridiculously simple, meant only to store information being manipulated by the application. The main entry bean will include its own functionality.

## Designing the entry bean

Each entry in the weblog system will be represented by a `BlogEntryBean`. Each entry has the following properties:

- `entryId`
- `blogId`
- `userId`
- `submitDate`
- `editDate`
- `entryTitle`
- `shortEntry`
- `completeEntry`
- `accesses`
- `feature`
- `active`
- `topicId`
- `topicName`

Each of these properties will be read-write properties, so we'll need to include both getter and setter methods. You'll also need to include methods that enable the bean to create and save new entries. These methods include:

- `loadEntry(int entryid)`

- saveEntry()
- deleteEntry()

But even with all of these methods defined, you're still one step away from actually building the bean.

## Testing the bean

Before you even think about building the bean itself, I recommend that you follow the Extreme Programming method of building your tests first. This may seem silly, but experience has shown me that it's not until I start building my tests that I realize I've forgotten a significant number of things. (I seem to consistently forget to build in the ability to delete objects, for example.)

While there are testing suites such as JUnit that will automate the testing process, I'm going to keep it simple and write a class that simply instantiates, manipulates, and displays the contents of our entries. You can then run the class every time you change something to make sure that nothing breaks and that the results come out as we expect them to.

The class itself is fairly straightforward:

```
import com.metrosphere.blog.*;

public class BlogEntryBeanTester {

    public static void main(String[] args) {

        System.out.println("Creating a new entry ...");
        BlogEntryBean bean = new BlogEntryBean();
        bean.setBlogId(100);
        bean.setUserId("nick");
        bean.setEntryTitle("Sample entry");
        bean.setShortEntry("This is the short entry.");
        bean.setCompleteEntry("This is the complete entry");
        bean.saveEntry();

        int newEntryId = bean.getEntryId();

        System.out.println("Reading the created entry ...");
        BlogEntryBean readBean = new BlogEntryBean();
        readBean.loadEntry(newEntryId);
        printEntry(readBean);

        System.out.println("Updating the entry ...");
        BlogEntryBean updateBean = new BlogEntryBean();
        updateBean.loadEntry(newEntryId);
        updateBean.setBlogId(100);
        updateBean.setUserId("sarah");
        updateBean.setEntryTitle("Sample entry edit");
        updateBean.setShortEntry("This is the edited short entry.");
        updateBean.setCompleteEntry("This is the edited complete entry");
        updateBean.setAccesses(updateBean.getAccesses()+1);
        updateBean.setActive(0);
        updateBean.setFeature(1);
```

```

        updateBean.saveEntry();
        printEntry(updateBean);

        System.out.println("Deleting the entry ...");
        updateBean.deleteEntry();
        printEntry(updateBean);

        ...
    }

    static void printEntry(BlogEntryBean bean) {
        System.out.println("EntryId = "+bean.getEntryId());
        System.out.println("BlogId = "+bean.getBlogId());
        System.out.println("UserId = "+bean.getUserId());
        if (bean.getSubmitDate() != null) {
            System.out.println("SubmitDate = "+bean.getSubmitDate().toString());
        } else {
            System.out.println("SubmitDate = null");
        }
        if (bean.getEditDate() != null) {
            System.out.println("EditDate = "+bean.getEditDate().toString());
        } else {
            System.out.println("EditDate = null");
        }
        System.out.println("EntryTitle = "+bean.getEntryTitle());
        System.out.println("ShortEntry = "+bean.getShortEntry());
        System.out.println("CompleteEntry = "+bean.getCompleteEntry());
        System.out.println("Accesses = "+bean.getAccesses());
        System.out.println("Feature = "+bean.getFeature());
        System.out.println("Active = "+bean.getActive());
        System.out.println("hasChanged = "+bean.getHasChanged());
        System.out.println("-----");
    }
}

```

Note that this is a *manual* test. You'll have to scan it visually to make sure that everything passed, as you'll see in [Test results](#).

## Creating the entry bean

Because this is a tutorial about JSP technology and not about Java, I won't go into detail about the construction of the `BlogEntryBean`, but I'll include it here for the curious. Notice that like the simple `TopicBean`, the `BlogEntryBean` has getter and setter methods for each of its properties, but that in some cases they're a bit more complex.

Notice also that wherever the properties are used within the class, they're used via their accessor methods. For example, in `saveEntry()`, rather than specifically referencing the `ENTRYTITLE` or `USERID` values, the bean uses the `getEntryTitle()` and `getUserId()` methods. This way the underlying implementation can change, but the methods that use them won't break.

```

package com.metrosphere.blog;

import java.util.Date;

```

```

import java.sql.*;

public class BlogEntryBean {

    boolean isNewEntry = true;
    boolean HASCHANGED = true;
    int ENTRYID = -1;
    int BLOGID = 1;
    String USERID;
    java.util.Date SUBMITDATE;
    java.util.Date EDITDATE;
    String ENTRYTITLE;
    String SHORTEENTRY;
    String COMPLETEENTRY;
    int ACCESSES = 0;
    int FEATURE = 0;
    int ACTIVE = 1;

    public void loadEntry (int entryid) {
        isNewEntry = false;
        Connection con = getConnection();
        try {
            Statement stmt = con.createStatement();
            ResultSet result = stmt.executeQuery(
                "select * from blogsystem.entries where entryid="+entryid);
            if (result.next()) {
                this.setEntryId(entryid);
                this.setBlogId(result.getInt("BLOGID"));
                this.setUserId(result.getString("USERID"));
                this.setSubmitDate(result.getTimestamp("SUBMITDATE"));
                this.setEditDate(result.getTimestamp("EDITDATE"));
                this.setEntryTitle(result.getString("ENTRYTITLE"));
                this.setShortEntry(result.getString("SHORTEENTRY"));
                this.setCompleteEntry(result.getString("COMPLETEENTRY"));
                this.setAccesses(result.getInt("ACCESSES"));
                this.setFeature(result.getInt("FEATURE"));
                this.setActive(result.getInt("ACTIVE"));
            } else {
                isNewEntry = true;
            }
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        this.setHasChanged(false);
    }

    public BlogEntryBean () {
        //New entry
    }

    public void setEntryId (int newEntryId){
        ENTRYID = newEntryId;
    }
    public void setBlogId (int newBlogId){
        BLOGID = newBlogId;
    }
    public void setUserId (String newUserId){
        USERID = newUserId;
    }

    /* These two properties are used only to clear the
    bean when the entry is deleted; otherwise, the database sets the date. */
    public void setSubmitDate (java.util.Date newSubmitDate){
        SUBMITDATE = newSubmitDate;
    }
    public void setEditDate (java.util.Date newEditDate){
        EDITDATE = newEditDate;
    }
}

```

```
public void setEntryTitle (String newEntryTitle){
    ENTRYTITLE = newEntryTitle;
    this.setHasChanged(true);
}
public void setShortEntry (String newShortEntry){
    SHORTEENTRY = newShortEntry;
    this.setHasChanged(true);
}
public void setCompleteEntry (String newCompleteEntry){
    COMPLETEENTRY = newCompleteEntry;
    this.setHasChanged(true);
}
public void setFeature (int newFeature){
    FEATURE = newFeature;
}
public void setAccesses (int newAccesses){
    ACSESSES = newAccesses;
}
public void setActive (int newActive){
    ACTIVE = newActive;
}
public void setHasChanged (boolean newHasChanged){
    HASCHANGED = newHasChanged;
}

public int getEntryId () {
    return ENTRYID;
}
public int getBlogId () {
    return BLOGID;
}
public String getUserId () {
    return USERID;
}
public java.util.Date getSubmitDate () {
    return SUBMITDATE;
}
public java.util.Date getEditDate () {
    return EDITDATE;
}
public String getEntryTitle () {
    return ENTRYTITLE;
}
public String getShortEntry () {
    return SHORTEENTRY;
}
public String getCompleteEntry () {
    return COMPLETEENTRY;
}
public int getFeature () {
    return FEATURE;
}
public int getAccesses () {
    return ACSESSES;
}
public int getActive () {
    return ACTIVE;
}
public boolean getHasChanged () {
    return HASCHANGED;
}

public void deleteEntry(int delEntryId){
    Connection con = getConnection();
    try {
        Statement stmt = con.createStatement();
        stmt.executeUpdate("delete from BLOGSYSTEM.ENTRIES where entryid = "
            +delEntryId);
    }
}
```

```

        stmt.close();
        con.close();
    } catch (java.sql.SQLException sqle) {
        sqle.printStackTrace();
    }
}

public void deleteEntry (){
    Connection con = getConnection();
    try {
        Statement stmt = con.createStatement();
        stmt.executeUpdate("delete from BLOGSYSTEM.ENTRIES where entryid = "
            +this.getEntryId());

        stmt.close();
        con.close();
    } catch (java.sql.SQLException sqle) {
        sqle.printStackTrace();
    }
    this.setEntryId(-1);
    this.setBlogId(-1);
    this.setUserId(null);
    this.setSubmitDate(null);
    this.setEditDate(null);
    this.setEntryTitle(null);
    this.setShortEntry(null);
    this.setCompleteEntry(null);
    this.setAccesses(-1);
    this.setFeature(-1);
    this.setActive(-1);
}

public void saveEntry() {
    Connection con = getConnection();
    String sql = null;
    try {
        PreparedStatement stmt = null;

        if (isNewEntry){
            //Get the next entry number
            Statement getSeqStmt = con.createStatement();
            ResultSet rs = getSeqStmt.executeQuery(
                "select NEXTVAL for BLOGSYSTEM.ID_SEQ from BLOGSYSTEM.DUMMY");
            if (rs.next()){
                this.setEntryId(rs.getInt(1));
            }
            rs.close();
            getSeqStmt.close();

            stmt = con.prepareStatement(
                "insert into BLOGSYSTEM.ENTRIES (blogid, userid, submitdate, "+
                "accesses, feature, active, entrytitle, shortentry, "+
                "completeentry, entryid) values (?,?,CURRENT TIMESTAMP, "+
                "?,?,,?,?,,?)");
            stmt.setString(6, this.getEntryTitle());
            stmt.setString(7, this.getShortEntry());
            stmt.setString(8, this.getCompleteEntry());
            stmt.setInt(9, this.getEntryId());
        } else if (this.getHasChanged()) {
            stmt = con.prepareStatement("update BLOGSYSTEM.ENTRIES set blogid = ?, "+
                "userid = ?, editdate = CURRENT TIMESTAMP, accesses = ?, "+
                "feature = ?, active = ?, entrytitle = ?, shortentry = ?, "+
                "completeentry = ? where entryid="+this.getEntryId());
            stmt.setString(6, this.getEntryTitle());
            stmt.setString(7, this.getShortEntry());
            stmt.setString(8, this.getCompleteEntry());
        } else {
            stmt = con.prepareStatement("update BLOGSYSTEM.ENTRIES set blogid = ?, "+
                "userid = ?, accesses = ?, feature = ?, active = ? where "+

```

```

        "entryid="+this.getEntryId());
    }

    stmt.setInt(1, this.getBlogId());
    stmt.setString(2, this.getUserId());
    stmt.setInt(3, this.getAccesses());
    stmt.setInt(4, this.getFeature());
    stmt.setInt(5, this.getActive());

    stmt.executeUpdate();

    stmt.close();
    con.close();

    this.loadEntry(this.getEntryId());
} catch (Exception e) {
    e.printStackTrace();
}
}

protected Connection getConnection(){
    Connection con = null;
    try {
        // register the DB2 JDBC driver with DriverManager
        Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
        String url = "jdbc:db2:blog";
        String username = "db2admin";
        String password = "db2password ";
        con = DriverManager.getConnection(url, username, password);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return con;
}
}
}

```

Once you build the bean, you can test it simply by running `BlogEntryBeanTester`.

## Test results

Running the `BlogEntryBeanTester` class against the newly created bean (hopefully) shows no errors. It does, however, require you to actually read the results -- or at least skim them -- to make sure everything is doing what you want it to. (See [Resources](#) for more information on automated testing.)

```

Creating a new entry ...
Reading the created entry ...
EntryId = 505
BlogId = 100
UserId = nick
SubmitDate = 2003-02-23 16:07:38.924001
EditDate = null
EntryTitle = Sample entry
ShortEntry = This is the short entry.
CompleteEntry = This is the complete entry

```

```
Accesses = 0
Feature = 0
Active = 1
Topic ID = 0
Topic Name = No Topic
hasChanged = false
-----
Updating the entry ...
EntryId = 505
BlogId = 100
UserId = sera
SubmitDate = 2003-02-23 16:07:38.924001
EditDate = 2003-02-23 16:07:41.147002
EntryTitle = Sample entry edit
ShortEntry = This is the edited short entry.
CompleteEntry = This is the edited complete entry
Accesses = 1
Feature = 1
Active = 0
Topic ID = 0
Topic Name = No Topic
hasChanged = false
-----
Deleting the entry ...
EntryId = -1
BlogId = -1
UserId = null
SubmitDate = null
EditDate = null
EntryTitle = null
ShortEntry = null
CompleteEntry = null
Accesses = -1
Feature = -1
Active = -1
Topic ID = -1
Topic Name = null
hasChanged = true
...
```

## Adding the bean to the page

OK, you've established that the bean works, so it's time to add it to the page. The actual process of adding it is virtually identical to the process you used with the simple `String` variable in the previous section, with one important exception:

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <% entryBean.loadEntry(553); %>
</jsp:useBean>
<html>
<head><title></title>
</head>
<body>

</body>
</html>
```

Whenever you instantiate a bean using the `<jsp:useBean>` action, any content that appears between the start tag and the end tag (`</jsp:useBean>`) is evaluated

by the server. If the content consists of text, the text is output. If the content consists of a scriptlet, as it does here, the commands in the scriptlet are executed. Here, I'm creating the new bean, calling it `entryBean`, and executing the `loadEntry()` method. I've arbitrarily set the entry to be loaded as number 553 because I know it exists; I created it at the end of the test run. In the production application, I'll be pulling this information from the request when the user clicks on a link.

## Using the bean on the page

Once you instantiate the bean, you can use it on the page just as you'd use any other Java object:

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <% entryBean.loadEntry(553); %>
</jsp:useBean>
<html>
<head><title></title></head>
<body style="font-family: Arial">

  <h1><%= entryBean.getEntryTitle() %></h1>
  <p style="font-size: smaller">
    <i>Posted on <%= entryBean.getSubmitDate() %>
      by <%= entryBean.getUserId() %>

      <% if (entryBean.getEditDate() != null) { %>
        <br />Edited on <%= entryBean.getEditDate() %>
      <% } %>
    </i>
  </p>

  <p><%= entryBean.getShortEntry() %></p>
  <p><%= entryBean.getCompleteEntry() %></p>

</body>
</html>
```

Here we're simply outputting the properties, as returned by their getter methods. The result is a page that shows the information that has been stored in the bean:

```
<html>
<head><title></title></head>
<body style="font-family: Arial">

  <h1>Building the system</h1>
  <p style="font-size: smaller">
    <i>Posted on 2003-02-20 16:56:25.735001
      by nick

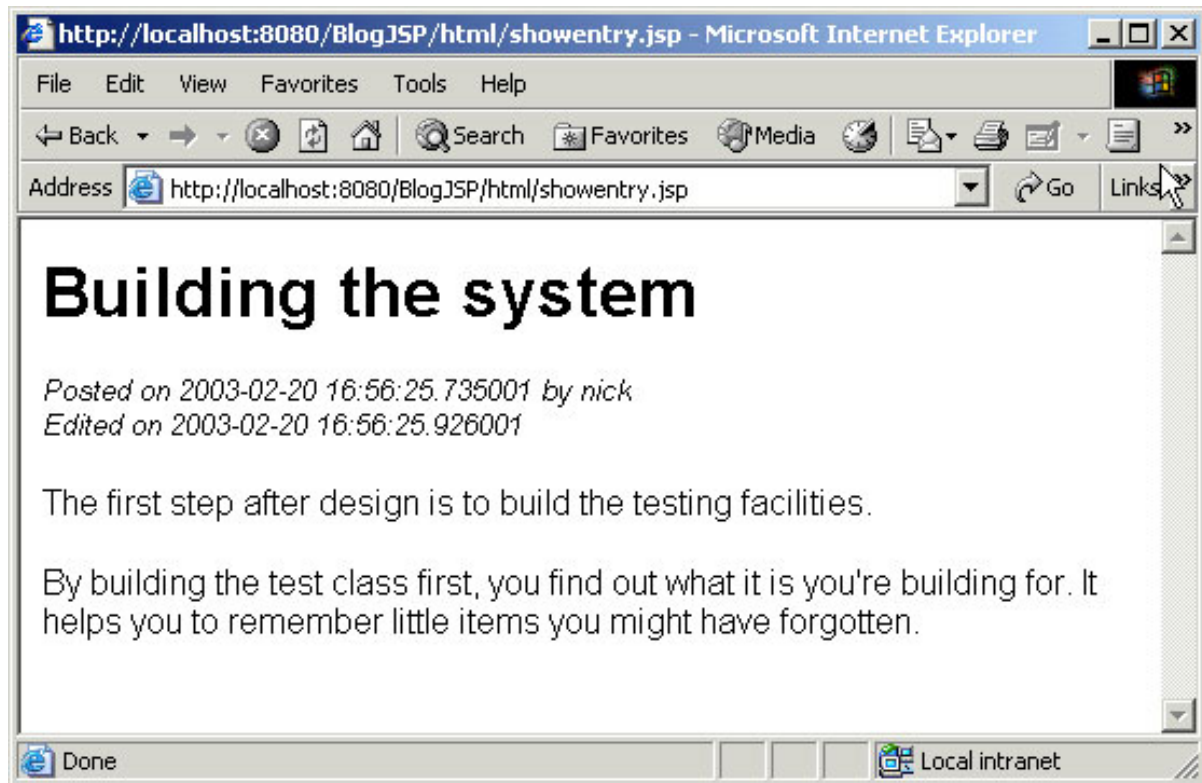
      <br />Edited on 2003-02-20 16:56:25.926001

    </i>
  </p>

  <p>The first step after design is to build the testing facilities.</p>
  <p>By building the test class first, you find out what it is you're building
  for. It helps you to remember little items you might have forgotten.</p>
```

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

The browser sees the page as normal HTML:



## Getting properties

Of course, while directly using the getter methods to retrieve the data does work, it doesn't exactly take advantage of all that JSP technology has to offer. Instead, you can retrieve a property by using the `jsp:getProperty` action, which takes the name of the object and the property we want to retrieve:

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">  
  <% entryBean.loadEntry(553); %>  
</jsp:useBean>  
<html>  
<head><title><jsp:getProperty name="entryBean" property="entryTitle"/></title></head>  
<body style="font-family: Arial">  
  
  <h1><%= entryBean.getEntryTitle() %></h1>  
  <p style="font-size: smaller">  
  ...
```

Using the action accomplishes the same thing as using the method directly, except that it's also automatically output without having to use the output shortcut.

```

<html>
<head><title>Building the system</title></head>
<body style="font-family: Arial">

    <h1>Building the system</h1>
    <p style="font-size: smaller">
...

```

## Setting properties

You can also use JSP actions to set properties, which might not seem like a big deal now, but will be really convenient when you look at [Setting parameters automatically](#). For example, you can set up the page so that every time you view an entry, the server increments the accesses property, then saves the new information:

```

<jsp:useBean id="entryBean" scope="request" class="com.metrosphere.blog.BlogEntryBean">
  <% entryBean.loadEntry(553); %>
</jsp:useBean>
<html>
<head><title><jsp:getProperty name="entryBean" property="entryTitle" /></title>

<jsp:setProperty name="entryBean" property="accesses"
  value="<%= entryBean.getAccesses()+1 %>" />
<% entryBean.saveEntry(); %>

</head>
<body style="font-family: Arial">

  <h1><%= entryBean.getEntryTitle() %></h1>
  <p style="font-size: smaller">
    <i>Posted on <%= entryBean.getSubmitDate() %>
      by <%= entryBean.getUserId() %>

      <% if (entryBean.getEditDate() != null) { %>
        <br />Edited on <%= entryBean.getEditDate() %>
      <% } %>
    </i>
  </p>

  <p><%= entryBean.getShortEntry() %></p>
  <p><%= entryBean.getCompleteEntry() %></p>

  <p style="font-size: 9pt">This entry has been accessed
  <jsp:getProperty name="entryBean" property="accesses" />
  times.</p>

</body>
</html>

```

Now, we've got a few things going on here, so let's take it one step at a time.

First we're instantiating and loading the bean, just as we did before, but before we do much of anything with it, we're setting a new value for the accesses property. (Notice that we're dynamically creating a value for the new property by mixing the two styles of JSP output.) After we set it, we're executing the `saveEntry()` method in order to save the new information, then proceeding as usual, outputting the new property value at the bottom of the page.

It's important to understand here that just because you're dealing with a Web page doesn't mean that you're disconnected from the rest of the system. When you tell the page to execute `saveEntry()`, the bean accesses the database to save the information.

This will come in handy in the next section, when you start working with the form.

---

## Section 5. Processing forms

### Creating the form

In this section, I talk about how HTML forms fit in with JavaServer Pages. If you're used to working with server-side technologies such as servlets, it may not be obvious how to process the information passed by a JSP, but it's actually pretty straightforward.

Start by creating a form that will be used to create a new blog entry:

```
<html>
<head>
<TITLE>Edit Blog Entry</TITLE>
</head>
<body>
<h1>Create a new entry</h1>
<form action="saveBlogEntry.jsp" method="post">
  <input type="hidden" name="userId" value="nick" />
  <input type="hidden" name="blogId" value="100" />
  <table>
    <tr><td>Title:</td><td><input type="text" name="entryTitle"
  /></td></tr>
    <tr>
      <td>Short Entry:</td>
      <td><textarea name="shortEntry" cols="50"
rows="3"></textarea></td>
    </tr>
    <tr>
      <td>Complete Entry:</td>
      <td><textarea name="completeEntry" cols="50"
rows="6"></textarea></td>
    </tr>
    <tr>
      <td></td>
      <td>
        <input type="submit" value="Save" />
        <input type="reset" value="Reset" />
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

Now, there's nothing particularly special about this page, which I'll call `editBlogEntry.jsp`. It's a simple HTML form, with the values for `userId` and `blogId` arbitrarily supplied. (In the eventual application, these values will come from the request itself, but that's not important right now.) The actual form fields have names that mimic the properties of the bean, but more of a convenience, as you will see, than a requirement. Otherwise, they can be arbitrarily set.

When the user submits the form, the information is sent to the `saveBlogEntry.jsp` page, which you'll create next.

## Using form parameters

JavaServer Pages were designed for the Web, so it makes sense that they would be optimized for the types of tasks that come up in the course of daily Web authoring. For example, in [Setting properties](#), you specifically set the value of a property, using the `value` attribute to set an arbitrary value. When submitting a form page, however, you have another option. The `jsp:setProperty` action can also look for a specific parameter value submitted as part of the request. For example, consider the `saveBlogEntry.jsp` page:

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <jsp:setProperty name="entryBean" property="blogId" param="blogId" />
  <jsp:setProperty name="entryBean" property="userId" param="userId" />
  <jsp:setProperty name="entryBean" property="entryTitle" param="entryTitle" />
  <jsp:setProperty name="entryBean" property="shortEntry" param="shortEntry" />
  <jsp:setProperty name="entryBean" property="completeEntry" param="completeEntry" />
  <% entryBean.saveEntry(); %>
</jsp:useBean>
<html>
<head><title>Your new entry</title>
</head>
<body>
  <p>Entry id: <%= entryBean.getEntryId() %><br />
  Blog id: <%= entryBean.getBlogId() %><br />
  <b><%= entryBean.getEntryTitle() %></b><br />
  <%= entryBean.getShortEntry() %><br />
  <%= entryBean.getCompleteEntry() %><br />
  Submitted by: <%= entryBean.getUserId() %><br />
  Submitted on: <%= entryBean.getSubmitDate() %><br />
  Edited on: <%= entryBean.getEditDate() %><br />
  Accessed <%= entryBean.getAccesses() %> times<br />
  Featured: <%= entryBean.getFeature() %><br />
  Active: <%= entryBean.getActive() %></p>
</body>
</html>
```

Here we're creating the bean, but we're not loading an entry, as we did before, because as of yet, there's no entry to load. Instead, we're setting the major properties of the bean by reading the parameters that were submitted by the form. Remember, we specifically named the form fields to mimic the bean properties; if we hadn't, we'd use the appropriate names as values for the `param` attribute.

After those properties have been set, we're saving the new entry, which also causes the bean to re-load itself. We can then display the properties of the bean. (Obviously this display is just utilitarian; in the final application, we'll pretty it up.)

In this way, you can take the information from a form, and save it to the database using the bean. But there's a specific reason you used the property names for the form fields, as you'll see next.

## Setting parameters automatically

There is, as you might have guessed, an easier way to access all of these properties. If you name the form fields after the properties of the bean, you can set them automatically using a wild card:

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <jsp:setProperty name="entryBean" property="*" />
  <% entryBean.saveEntry(); %>
</jsp:useBean>
<html>
<head><title>Your new entry</title>
</head>
<body>
  <p>Entry id: <%= entryBean.getId() %><br />
  ...
```

In this case, the server automatically looks for a parameter that matches each property for the bean, and if it finds one, it sets the value for that property. One of the nice things about this way of doing things is that you can change the form page without changing the form submission, and things should still work -- as long as everything is named properly.

## Adding editing capabilities

Now you have a way to create new entries, but once they're created, you're still going to need a way to edit them. The best way to do that is to modify your `editBlogEntry.jsp` page so that if it's being called for an existing bean, it will pre-populate the form. If not, it will simply display an empty form for the user to complete.

This page also uses a new construction: the declaration block.

```
        <%!
int oldEntryId = -1;
int oldBlogId = -1;
String oldUserId = "nick";
String oldEntryTitle = "";
String oldShortEntry = "";
String oldCompleteEntry = "";
```

```

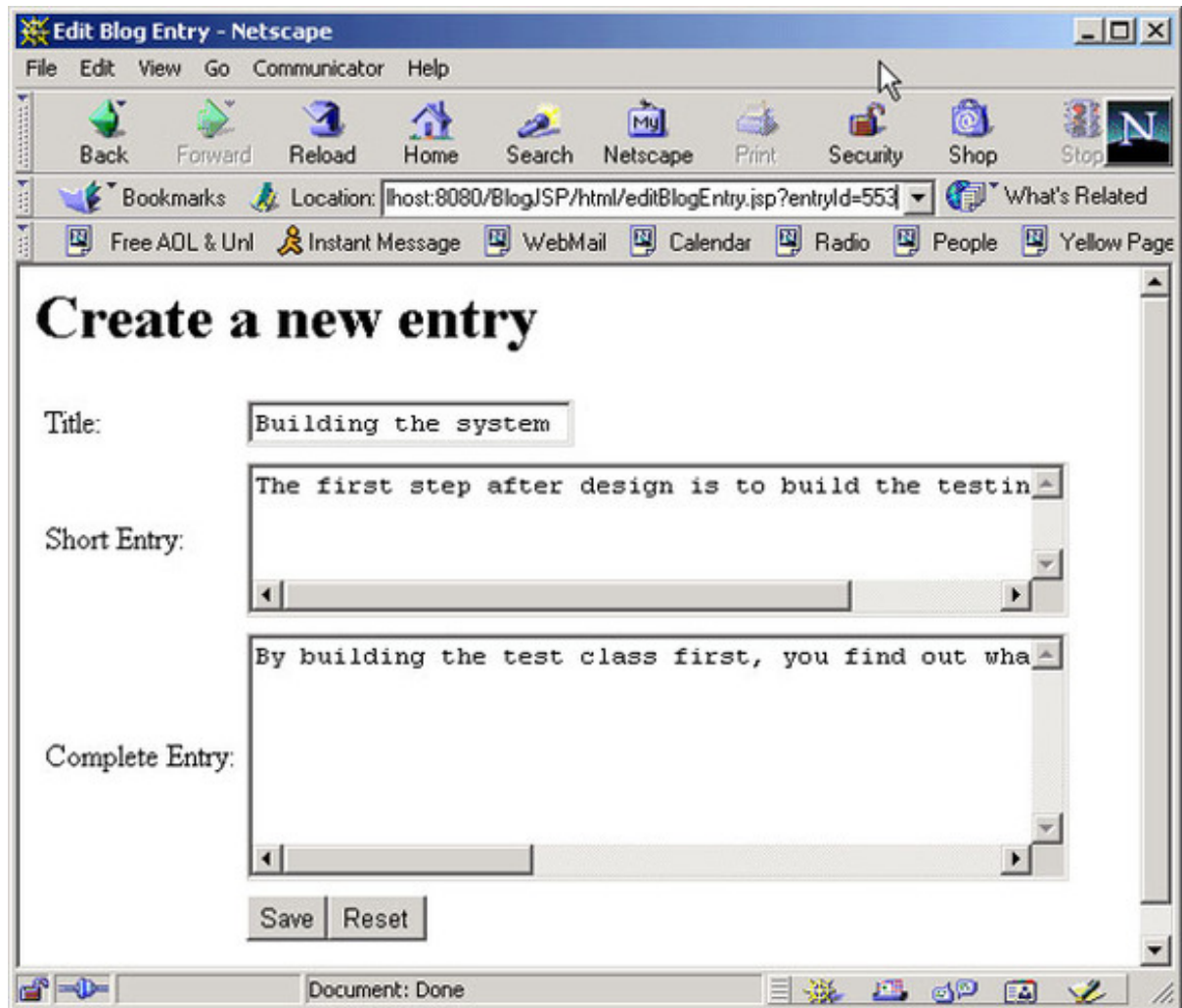
%>
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <jsp:setProperty name="entryBean" property="*" />
  <%
    if (entryBean.getEntryId() > 0) {
      entryBean.loadEntry(entryBean.getEntryId());
      oldEntryId = entryBean.getEntryId();
      oldBlogId = entryBean.getBlogId();
      oldUserId = entryBean.getUserId();
      oldEntryTitle = entryBean.getEntryTitle();
      oldShortEntry = entryBean.getShortEntry();
      oldCompleteEntry = entryBean.getCompleteEntry();
    }
  %>
</jsp:useBean>
<html>
<head>
<TITLE>Edit Blog Entry</TITLE>
</head>
<body>
<h1>Create a new entry</h1>
<form action="saveBlogEntry.jsp" method="post">
  <input type="hidden" name="entryId" value="<%= oldEntryId %>" />
  <input type="hidden" name="userId" value="<%= oldUserId %>" />
  <input type="hidden" name="blogId" value="<%= oldBlogId %>" />
  <table>
    <tr>
      <td>Title:</td>
      <td><input type="text" name="entryTitle"
        value="<%= oldEntryTitle %>" /></td>
    </tr>
    <tr>
      <td>Short Entry:</td>
      <td><textarea name="shortEntry" cols="50"
        rows="3"><%= oldShortEntry %></textarea></td>
    </tr>
    <tr>
      <td>Complete Entry:</td>
      <td><textarea name="completeEntry" cols="50"
        rows="6"><%= oldCompleteEntry %></textarea></td>
    </tr>
    <tr>
      <td align="right"></td>
      <td>
        <input type="submit" value="Save" />
        <input type="reset" value="Reset" />
      </td>
    </tr>
  </table>
</form>
</body>
</html>

```

The first section, delimited by `<%! %>`, is a JSP declaration block, used only to declare variables that we'll use later. From there, we're creating a new bean, which automatically sets the `entryId` property to `-1`. Next we're setting any properties that were submitted with the page. If an `entryId` parameter had been set, the `entryId` property for the bean will now be equal to that value, which will signal to the server that it should load that entry. If that's the case, the variables declared in the opening block will be set to the current values for the entry. If not, the variables will remain in their default state.

In either case, they'll be used to populate the form. If there was no entry to load, the

user will be none the wiser, but the new entryId field will be set to -1, signaling to saveEntryBean.jsp that this is a new entry.



## Saving the changes

Because of the way we constructed the page, the `saveBlogEntry.jsp` page actually requires very little modification to accommodate the changes we made to `editBlogEntry.jsp`. All we need to do is check the value of the `entryId` parameter. If it's not `-1`, we have to load the current values for the entry. From there, everything is the same; we set any properties that were submitted, and then save the bean in its new state.

```
<jsp:useBean id="entryBean" scope="page" class="com.metrosphere.blog.BlogEntryBean">
  <% if (!request.getParameter("entryId").equals("-1")) {
    Integer submittedInteger = new Integer(request.getParameter("entryId"));
    entryBean.loadEntry(submittedInteger.intValue());
  }
  %>
</jsp:useBean>
```

```

    %>
    <jsp:setProperty name="entryBean" property="*" />
    <% entryBean.saveEntry(); %>
</jsp:useBean>
<html>
<head><title>Your new entry</title>
</head>
<body>
  <p>Entry id: <%= entryBean.getEntryId() %><br />
  Blog id: <%= entryBean.getBlogId() %><br />
  <b><%= entryBean.getEntryTitle() %></b><br />
  <%= entryBean.getShortEntry() %><br />
  <%= entryBean.getCompleteEntry() %><br />
  Submitted by: <%= entryBean.getUserId() %><br />
  Submitted on: <%= entryBean.getSubmitDate() %><br />
  Edited on: <%= entryBean.getEditDate() %><br />
  Accessed <%= entryBean.getAccesses() %> times<br />
  Featured: <%= entryBean.getFeature() %><br />
  Active: <%= entryBean.getActive() %></p>
</body>
</html>

```

That covers the basics. The next section explains how to take all of this one step further and create a custom JSP tag library.

---

## Section 6. Tag libraries

### Tag library basics

A tag library, to put it simply, is a way of including functionality on a JSP page without having to include the actual code. Consider, for example, this version of the WebSphere Portal navigation bar, modified slightly. (The sections delineated by `<%--` and `--%>` are comments.)

```

<%-- Licensed Materials - Property of IBM, 5724-B88, (C) Copyright IBM
Corp. 2001, 2002 -
All Rights reserved. --%>

<table class="wpsNavbar" width="150" border="0" cellspacing="0"
cellpadding="0">

  <wps:pageLoopInit/>
  <wps:pageLoop>

    <wps:if pageSelected="yes">
      <%-- selected page tab --%>
      <tr>
        <td valign="middle" class="wpsSelectedTab">
          <table width="100%" border="0" cellspacing="8"
cellpadding="0">
            <tr>
              <td class="wpsSelectedTab"><a
href='<wps:urlParent/>'

```

```

                                class="wpsSelectedTab">
                                <wps:page attribute="title"/></a></td>
                                </tr>
                                </table>
                                </td>
                                </tr>
                                </wps:if>

                                <wps:if pageSelected="no">
                                <!-- non-selected page tab --%>
                                <tr>
                                <td valign="middle" class="wpsTabs">
                                <table width="100%" border="0" cellspacing="8"
                                cellpadding="0">
                                <tr>
                                <td class="wpsTabs"><a href='<wps:urlParent/>'
                                class="wpsTabs">
                                <wps:page attribute="title"/></a></td>
                                </tr>
                                </table>
                                </td>
                                </tr>
                                </wps:if>

                                <tr>
                                <td class="wpsNavbarSeparator" width="100%" height="1">
                                <img width="1" height="1" alt=""
                                src='<%= wpsBaseUrl %>/images/dot.gif'
                                border="0"></td>
                                </tr>

                                </wps:pageLoop>

                                <!-- This is here so Netscape 4.x will work properly --%>
                                <tr>
                                <td width="150"><img width="150" alt=""
                                src='<wps:urlFindInTheme file="navfade.jpg"/>'
                                border="0"></td>
                                </tr>

                                </table>

```

This page doesn't actually appear on its own on the portal. Instead, it's included in another page, which includes the definition:

```
<%@ taglib uri="/WEB-INF/tld/engine.tld" prefix="wps" %>
```

This tells the server that when it encounters a tag that starts with the `wps:` prefix, it should check the file `engine.tld` to find out what to do with it. The `engine.tld`, or Tag Library Definition file, defines the Java classes that correspond to each of these tags. For example, when the server encounters the `<wps:if pageSelected="yes">` tag, it knows to consult a particular Java class in order to determine whether to display the content within that tag.

In this section, you're going to create a simple tag library that helps manage the display of a list of topics on the blog entry page.

## Adding the tag to the page

First you'll add the tag to the page, and then work your way back through the process until we get to the class itself. I'll start with a very simple tag that outputs the HTML for a select list of topic names:

```

                                <%@ taglib uri="/taglib.tld" prefix="blogutil" %>
<%!
    int oldEntryId = -1;
...
<h1>Create a new entry</h1>
<form action="saveBlogEntry.jsp" method="post">
    <input type="hidden" name="entryId" value="<%= oldEntryId %>" />
    <input type="hidden" name="userId" value="<%= oldUserId %>" />
    <input type="hidden" name="blogId" value="<%= oldBlogId %>" />
    <table>
        <tr><td>Topic:</td><td><bblogutil:topiclist/></td></tr>
        <tr>
            <td>Title:</td>
            <td><input type="text" name="entryTitle"
                value="<%= oldEntryTitle %>" /></td>
        </tr>
    ...

```

Here I've defined the `blogutil:` prefix and specified that it refers to the `/taglib.tld` tag library definition. I've then referenced the `topiclist` tag that's part of that definition. Ultimately, that tag should be replaced by a `select` box such as:

```

...
    <tr><td>Topic:</td><td>
        <select name="topicId">
            <option value="0" >No Topic</option>
            <option value="1" >Java</option>
            <option value="2" >Databases</option>
            <option value="3" >Miscellaneous</option>
        </select>
    </td></tr>
...

```

The first step will be to tell the server where to find `/taglib.tld`.

## Referencing the library in web.xml

Every J2EE Web application has an associated `web.xml` file that determines such values as the application's default files, any defined servlets, and so on. The `web.xml` file also stores the actual location of your tag library files along with the URL alias that refers to them. For example, the `web.xml` file below indicates that

the URL `/taglib.tld` actually refers to the file `taglib.tld` in the `/WEB-INF` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app id="WebApp">
  <display-name>BlogJSP</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <taglib>
    <taglib-uri>/taglib.tld</taglib-uri>
    <taglib-location>/WEB-INF/taglib.tld</taglib-location>
  </taglib>
</web-app>
```

Next we'll look at the `.tld` file itself.

## Creating the library

The main purpose of the tag library definition file is to relate the tag name back to the class that implements it. Here we see a simple `.tld` file that indicates that the `topiclist` tag actually refers to the `TopicListTag` class.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname></shortname>
  <tag>
    <name>topiclist</name>
    <tagclass>com.metrosphere.util.TopicListTag</tagclass>
  </tag>
</taglib>
```

Now let's look at the class itself.

## The class

From the discussion so far, it may seem like any Java class can be turned into a tag, but that's not actually the case. In order to be used as a tag, a class must implement particular interfaces -- or extend support classes, as we're doing here -- and it must implement particular methods.

The example we have so far is a simple element; it has no body, just a single tag that acts as both start tag and end tag. For this we can create a class that extends the `javax.servlet.jsp.tagext.TagSupport` class. When the server encounters the `topiclist` tag, it executes the `doStartTag()` method.

```

package com.metrosphere.util;

import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
import java.sql.*;

public class TopicListTag extends TagSupport {

    public TopicListTag() {
        super();
    }

    public int doStartTag() {
        String output = generateTopicList();
        try{
            javax.servlet.jsp.JspWriter out = pageContext.getOut();
            out.println(output);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return SKIP_BODY;
    }

    public static String generateTopicList() {
        String topicString = "<select name=\"topicId\">";
        Connection con = getConnection();
        try {
            Statement stmt = con.createStatement();
            ResultSet result = stmt.executeQuery("select * from blogsystem.topics");
            while (result.next()) {
                topicString = topicString + "<option value=\"";
                topicString = topicString + result.getInt("topicid");
                topicString = topicString + "\">";
                topicString = topicString + result.getString("topicname");
                topicString = topicString + "</option>";
            }
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        topicString = topicString + "</select>";
        return topicString;
    }

    protected static Connection getConnection(){

        Connection con = null;
        try {
            // register the DB2 JDBC driver with DriverManager
            Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
            String url = "jdbc:db2:blog";
            String username = "db2admin";
            String password = "db2password ";
            con = DriverManager.getConnection(url, username, password);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return con;
    }
}

```

```
}  
}
```

The `doStartTag()` method simply retrieves a string and prints it to the page using the `JspWriter` object obtained from the `pageContext`. The `pageContext` itself is built into the `TagSupport` class.

It's a bit of a circuitous route, but we end up with the HTML text output to the page when the server encounters the `blogutil:topiclist` tag.

## Adding an attribute

Now let's make things just a little more complex. The `topiclist` tag as it exists right now is going to output the same text no matter where you put it, no matter what the situation. But I want to be able to tell it what `topicId` should be shown as selected so that it's useful when I'm editing an existing entry, and not just when I'm creating a new one. You can do that by adding an attribute to the tag:

```
...  
    <tr>  
        <td>Topic:</td>  
        <td><bblogutil:topiclist selected="<%= oldTopicId %>" /></td>  
    </tr>  
...
```

The tag is the same, but you've added a bit more information, which just happens to be set dynamically based on the value of `oldTopicId`. In order for the server to know that it needs to send that attribute to the class, you need to add the attribute to the tag library definition:

```
...  
<tag>  
  <name>topiclist</name>  
  <tagclass>com.metrosphere.util.TopicListTag</tagclass>  
  <attribute>  
    <name>selected</name>  
    <required>>false</required>  
    <rteprvalue>>true</rteprvalue>  
  </attribute>  
</tag>  
...
```

Here you're specifying that the `selected` attribute is optional, and that if it is present, it can be determined by an expression at run time.

Now you need to add the attribute to the class.

## Reading the attribute

Adding the attribute to the class is actually very simple. All you need to do is add it as a property and include a getter and setter method:

```
package com.metrosphere.util;

import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.tagext.TagData;
import java.sql.*;

public class TopicListTag extends TagSupport {

    public TopicListTag() {
        super();
    }

    private int selected;
    public void setSelected(int newSelected) {
        selected = newSelected;
    }
    public int getSelected() {
        return selected;
    }

    public int doStartTag() {

        String output = generateTopicList(selected);
        try{
            javax.servlet.jsp.JspWriter out = pageContext.getOut();
            out.println(output);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return SKIP_BODY;
    }

    public static String generateTopicList(int isSelected) {
        String topicString = "<select name=\"topicId\">";
        Connection con = getConnection();
        try {
            Statement stmt = con.createStatement();
            ResultSet result = stmt.executeQuery("select * from blogsystem.topics");
            while (result.next()) {
                topicString = topicString + "<option value=\"";
                topicString = topicString + result.getInt("topicid");
                topicString = topicString + "\"";
                if (isSelected == result.getInt("topicid")) {
                    topicString = topicString + " selected=\"selected\" ";
                }
                topicString = topicString + ">";
                topicString = topicString + result.getString("topicname");
                topicString = topicString + "</option>";
            }
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        topicString = topicString + "</select>";
        return topicString;
    }
    ...
}
```

Notice that you are accessing the value of `selected` directly. Notice specifically that you didn't have to do anything special to set that value. The server took care of it when the class was called. From there, you're simply using it to affect the way that `generateTopicList()` creates the `String`.

## Adding body content

You may have noticed that in the Portal example in [Tag library basics](#), the content of custom tags figures heavily into the picture. The `wps:if` element determines whether to output the contents of the body, and the `wps:pageLoop` element repeatedly evaluates its content, making decisions along the way. You're going to add some of these capabilities to your tag library by changing `topiclist` so that for each topic, it outputs the body of the tag, inserting the appropriate information:

```
...
    <tr><td>Topic:</td><td>
        <select name="topicId">
            <blogutil:topiclist selected="<%= oldTopicId %>">
                <option value='<jsp:getProperty name="topicelement"
                    property="topicId" />'
                    <jsp:getProperty name="topicelement" property="selected" />>
                <jsp:getProperty name="topicelement" property="topicName" />
            </blogutil:topiclist>
        </select>
    </td></tr>
...
```

The advantage of doing it this way is that the `topiclist` can be used for any type of arrangement, from a simple list of topics to a select list of checkboxes. The JSP page can be changed, and the tag will simply output the data in the appropriate places.

## Iterating and evaluating body content

To make the class recognize and evaluate the body, you need to make significant changes.

First, you need to extend the `BodyTagSupport` class rather than `TagSupport` class. Second, you need to add two more methods.

```
package com.metrosphere.util;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.TagData;
import java.sql.*;

public class TopicListTag extends BodyTagSupport {

    public TopicListTag() {
        super();
    }

    private int selected;
    public void setSelected(int newSelected) {
        selected = newSelected;
    }
}
```

```

public int getSelected() {
    return selected;
}

protected static Connection getConnection(){
    ...
}

Connection con = getConnection();
ResultSet result = null;

public int doStartTag() {

    try {
        Statement stmt = con.createStatement();
        result =
            stmt.executeQuery("select * from blogsystem.topics order by topicid");

        if (result.next()) {
            TopicBean topic = new TopicBean();
            topic.setTopicId(result.getInt("topicid"));
            topic.setTopicName(result.getString("topicname"));
            if (selected == result.getInt("topicid")){
                topic.setSelected(" selected=\\"selected\\" ");
            }
            pageContext.setAttribute("topicelement", topic);
            return EVAL_BODY_TAG;
        } else {
            return SKIP_BODY;
        }

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

public int doAfterBody() throws JspException {

    try {

        try {
            bodyContent.writeOut(bodyContent.getEnclosingWriter());
            bodyContent.clear();
        } catch (java.io.IOException e) {
            e.printStackTrace();
        }

        if (result.next()) {
            TopicBean topic = new TopicBean();
            topic.setTopicId(result.getInt("topicid"));
            topic.setTopicName(result.getString("topicname"));
            if (selected == result.getInt("topicid")){
                topic.setSelected(" selected=\\"selected\\" ");
            }
            pageContext.setAttribute("topicelement", topic);

            return EVAL_BODY_TAG;
        } else {
            return SKIP_BODY;
        }

    } catch (SQLException sqle) {
        sqle.printStackTrace();
        return SKIP_BODY;
    }
}

public int doEndTag() {

```

```
        try{
            result.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return EVAL_PAGE;
    }
}
```

Here it helps to understand two things. First, the constants, `EVAL_BODY_TAG`, and `SKIP_BODY`, control what the class does next. Second, the methods are executed in a particular order. Let's take it step-by-step.

First, when the server encounters the start tag, it executes the `doStartTag()` method. This method initializes the `ResultSet`, and if any records are found, it instantiates and populates a `TopicBean`. (The `TopicBean` is the same class that we talked about in [What is a bean?](#), with the addition of a `String` selected property.) Once the bean is populated, the real magic happens.

The line

```
pageContext.setAttribute("topicelement", topic);
```

creates an object named `topicelement` and populates it with the `topic` object. Essentially, it's as if you had the line

```
<jsp:useBean id="topicelement" class="com.metrosphere.util.TopicBean" />
```

so on the page, when you call

```
<jsp:getProperty name="topicelement" property="topicName" />
```

the server outputs the properties that were set within the class.

Once the new bean has been associated with the page, returning the `EVAL_BODY_TAG` tells the class to go on evaluating the body content by moving on, in this case to the `doAfterBody()` method. (If there are no topics, it returns `SKIP_BODY`, which sends it directly to the `doEndTag()` method.)

When the `doStartTag()` method returned `EVAL_BODY_TAG`, that was the class's cue to create a `bodyContent` object out of the content of the `topiclist` tag. The `doAfterBody()` method then takes that `bodyContent` object and outputs it as it stands at this moment. It then clears the `bodyContent` object so you can load it up again, which you'll do by checking from the next record. If there is one, you'll once again associate the `TopicBean` with the body, and you'll create a new `bodyContent` object by returning `EVAL_BODY_TAG`. This also causes the class to

execute the `doAfterBody()` method again, outputting the current version of the `bodyContent`.

Eventually, you'll run out of topics and return `SKIP_BODY`, which causes the class to move on to `doEndTag()`. The `doEndTag()` method closes the database objects and returns `EVAL_PAGE`, which causes the server to continue outputting the rest of the JSP page.

---

## Section 7. Summary

### What you've learned

At this point, you've got enough information to create a JavaServer Pages application. You've learned how to create objects within a page and to read and set their properties. You've also learned how to add logic to the page, and how to create and use custom tag libraries. The tutorial covered:

- JSP basics
- Creating the weblog entry objects
- Using properties
- Using forms with JavaServer Pages
- Using custom tags
- Creating custom tags

I've just covered the basics here. You have all of the information you need to create most JSP applications, but there are lots of details you should familiarize yourself with. Check out the resources section to get more information.

# Resources

## Learn

- Get up to speed on the MetroSphere project by checking out the [Series overview](#), which lists all the articles in this series.
- Get more information about [WebSphere Portal](#) and check out WebSphere zone [developer resources](#).
- Get an overview of current Portal product offerings, and an explanation of what each offering provides. Check out [WebSphere Portal Product Offerings](#).
- To get a feel for the way things work together, read about "[JSP Architecture](#)" (*developerWorks*, February 2001).
- To get a more in-depth look at JavaServer Pages, take the [Introduction to JavaServer Pages technology](#) tutorial. It provides more of a reference guide, showing all of the different possibilities.
- Get another look at custom libraries with "[Taking control of your JSP pages with custom tags](#)" (*developerWorks*, January 2002) and "[JSP Taglibs: better usability by design](#)" (*developerWorks*, December 2001).
- Learn more about "[Dynamic Web-based data access using JSP and JDBC technologies](#)" (*developerWorks*, September 2001).
- Check out "[Ten JSP technology books compared](#)" (*developerWorks*, June 2001).
- Read about automated unit testing using JUnit, which enables you to automatically run tests without having to manually scan the results, in [Test Infected](#), from sourceforge.com.

## Get products and technologies

- Download trial version of [DB2 Universal Database](#).
- Download trial versions of [other IBM software products](#).

## Discuss

- [Participate in the discussion forum for this content](#).

# About the author

## Nicholas Chase

Nicholas Chase has been involved in Web-site development for companies such as Lucent Technologies, Sun Microsystems, Oracle,

and the Tampa Bay Buccaneers. He has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams).