Building a J2EE application with Domino and WebSphere

More than the sum of their parts

Skill Level: Introductory

David Gallardo (David@Gallardo.org)
Software Consultant

24 Jul 2002

Lotus Domino and WebSphere Application Server are both platforms for building distributed, server-based applications. They have different strengths: Application Server provides a complete J2EE platform while Domino provides the unique ability to build collaborative applications. After briefly surveying the various possible ways the two can work together using Java, the tutorial concentrates on how Domino can be used in an Application Server environment using standard multi-tier J2EE design. It pays special attention to the issue of separating presentation logic from business logic and how Domino can participate in the Model-View-Controller (MVC) design using JavaServer Pages (JSP), Java servlets and Enterprise JavaBeans (EJB).

Section 1. Introduction

Should I take this tutorial?

This tutorial is for developers who want to use Lotus Domino's collaborative features in the context of a standard Java 2 Enterprise Edition (J2EE) environment. It covers essential setup and configuration issues but concentrates on how to design and code with the Domino classes in JavaServer Pages (JSPs), servlets, and Enterprise JavaBeans (EJBs).

It is recommended that readers be familiar with the following technologies:

- JSP pages and EJBs

- Domino, especially programming with the Domino classes

See Tutorial resources for more information on these topics.

What is this tutorial about?

Lotus Domino and WebSphere Application Server are both platforms for building distributed, server-based applications. They have different strengths: Application Server provides a complete J2EE platform while Domino provides the unique ability to build collaborative applications. The tutorial begins by briefly surveying the various possible ways the two can work together using Java. It then concentrates on how Domino can be used in an Application Server environment using standard multi-tier J2EE design. It pays special attention to the issue of separating presentation logic from business logic and how Domino can participate in the Model-View-Controller (MVC) design pattern using JSP pages, Java servlets and EJBs.

The tutorial uses a sample application that displays the contents of a Lotus Domino discussion database to demonstrate how Domino can work with WebSphere Application Server and J2EE. This will take us from the nuts and bolts of configuring Domino and Application Server to the more abstract realm of designing and building a J2EE application using the MVC methodology.

What the sample application does not cover

The sample application is intended to demonstrate simply and clearly how Domino can fit into the J2EE architecture. It is not intended to be a complete demonstration of J2EE. Specifically, key features of Application Server that make it a compelling platform for distributed application development -- especially EJB and relational database connectivity -- are not demonstrated because these topics would require coverage well beyond the scope of this tutorial.

The sample application might be, for example, part of an online music store where features such as the catalog and transaction processing are built using standard J2EE components such as JSP pages, servlets, EJBs, and a relational database. Although the Domino part can easily be implemented using Domino's Web publishing features, calling the Domino Java classes from a J2EE component, such as a servlet as shown here, provides greater flexibility and control over the application's appearance and behavior, and makes it easier to achieve a seamless integration.

For an example of a larger application, see the IBM Redbook, *Domino and WebSphere Together Second Edition* , available online (see Tutorial resources). This book provides thorough coverage of integrating Domino, Application Server and DB2 Universal Database using servlets, JSP pages and EJB technology.

Tools and configuration recommendations

In order to follow along with this tutorial, you'll need the following applications installed. Free trial versions are available for all three:

- WebSphere Application Server. You can download a free trial of the latest version of Application Developer.
- WebSphere Studio Application Developer. This tutorial uses version 4.0.3. You can download a free trial of the latest version of Application Developer.
- Lotus Notes and Domino 6 (See the Lotus Developer Domain for details on Notes/Domino 6.)

The tutorial assumes that the Application Developer configuration of WebSphere Studio, WebSphere Application Server, Advanced Edition and Lotus Domino have been installed with default options, preferably, but not necessarily, on the same computer. The instructions for building the sample application have been tested with Application Developer V4.0.3, Application Server V4.0.1 and V3.5, and Domino 6.0. The sample application was built, tested, and deployed on Windows, but will work equally well on Linux.

You'll also need to download DominoWASsrc.zip, the source code for the sample application.

A Pentium® class PC running Windows NT/2000 with a clock speed of at least 500 MHz and at least 256 MB of memory is recommended. The examples here ran well on a 650 MHz AMD Athlon™-based system with 384 MB of memory. Hard disk space requirements for both Application Server and Domino are approximately 1 GB after installation; downloading from the Web and installing from the same hard disk will temporarily require approximately 1 GB of additional space.

Section 2. Domino and WebSphere Application Server

Overview

Lotus Domino and WebSphere Application Server are both platforms for building enterprise-wide distributed applications. They are both rich in features and have mutual support for a broad range of technologies and protocols. While there is some overlap in functionality -- particularly in their support for HTTP, HTML and Java -- the many ways in which they can interoperate ultimately makes for a powerful combination. This section examines their relative strengths and recommends ways that they can be most advantageously integrated.

Key strengths: Domino

Both Lotus Domino and WebSphere Application Server have a long list of features intended to support similar goals, but despite some overlap in secondary features, they are remarkably different in their key strengths:

Domino allows building enterprise applications easily and quickly using a hierarchical, document-based approach. In addition to its native tools and environment (Notes client support, tools, agents, a Visual Basic-like programming language, LotusScript, and so on), Domino also provides support for:

- Publishing its databases to the Web as HTML using the integrated Domino HTTP server
- Java servlets in the HTTP server (Domino 6.0 provides a JSP tag library where the JSP pages you develop will execute on WebSphere Application Server)
- Java as a back-end programming language for applications, applets, and agents
- Directory Services (LDAP)

Key strengths: Application Server

WebSphere Application Server is the premier J2EE implementation from IBM. As such, it provides a platform for developing Web-based, multi-tier, component-based applications that are scalable, robust, and secure and based on standard Java technologies. The key features from a development point of view are its support for:

- Java servlets
- JSP pages
- EJBs
- Legacy applications and databases

Domino and Application Server working together

While there are many different ways that Lotus Domino and WebSphere Application Server can communicate and interoperate, doing this in a way that makes sense is critical.

The first question that needs to be answered is: Why use Domino and Application Server together in one application? There are three possible answers, depending on your starting point:

1. If you have an existing Domino application, you may wish to extend it or refactor it using J2EE technologies to:

- Use standard, well-understood, multi-tier J2EE design patterns such as the Model-View-Controller (MVC) architecture. This can improve scalability, maintainability, extensibility, and interoperability with other distributed applications and resources.
- Use specific J2EE technologies such as EJBs, transactions, and support for legacy applications for integration into existing systems.

2. If you already have a J2EE application using Application Server you may wish to integrate with Domino to:

- Use data in an existing Domino application.
- Add new features or resources, such as a document or discussion database, for which Domino is ideally suited.

3. If you are starting from scratch, you may find that Domino offers unique and compelling features such as distributed content development, messaging and workflow, and directory and security services. At the same time, you may have existing applications using J2EE, or requirements demanding support for J2EE as the central technology.

In each of these cases, the goal is to use Domino together with Application Server to develop applications featuring the key benefits of both: a collaborative application that is scalable, robust, secure, and based on standard Java technology.

Section 3. Configuration for collaboration

Domino integration features

As suggested in the previous section, there are a number of features in Lotus Domino and WebSphere Application Server that can be used to share information. The Java programming language and its associated technologies are the key to this integration.

Domino is not a Java platform, but it provides support for Java throughout. This support, based on a Java API that wraps the native C++ Domino classes, allows writing:

- Java agents -- Java programs that can, for example, run automatically at scheduled times to update a database.
- Java servlets -- Java programs that are run automatically by the Domino Web server in response to a request from a client browser.

- Java applications -- Applications that run outside the server.
- JDBC driver -- A package that allows us to access Domino databases as though they were relational databases.

Of these, Java applications and a JDBC driver are those most relevant for integration with Application Server. Because they enable you to write Domino applications that run outside the context of the Domino server, they provide a way to extend Domino by calling it from a J2EE context.

Additional Domino features

In addition to Lotus Domino's Java features, there are two other features that are of special interest for integration purposes:

- Directory services -- Domino provides directory services, specifically an implementation of the Lightweight Directory Access Protocol (LDAP) which can be used for user authentication in Application Server.
- The Domino Web Server API -- Domino enables third-party software to extend its Web server using the Domino Web Server API (DSAPI). Application Server, notably, provides a DSAPI plug-in that allows it to work with the Domino Web server, rather that the IBM HTTP server with which it is bundled.

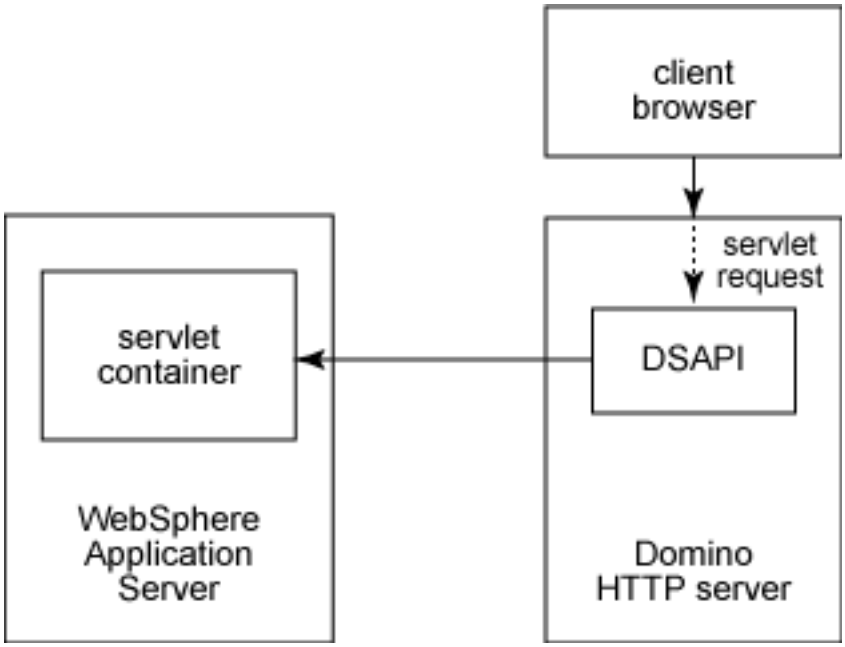
Configuring Lotus Domino

There are three choices to make before configuring Lotus Domino:

- Will Lotus Domino and WebSphere Application Server be on the same machine? As already noted, this tutorial assumes that they will be on the same machine, but will note when something needs to be done differently if they are not.
- Which Web server should you use? Domino has one and Application Server comes with the IBM HTTP Server. For development purposes, it is convenient to set up Application Server and Domino on the same machine. In order to use LDAP authentication, however, the Domino Web server *must* be running on port 80. Given that, it doesn't seem worthwhile to run the IBM server as well, so the example here will use the Domino Web server.
- Should you use Domino or Application Server for servlets? Given Domino's lack of support for JSP pages (as of Release 5) and Application Server's better deployment capabilities, Application Server is preferable and will be used here. This also makes using EJBs easier.

The basic Web server configuration

The following illustration provides an overview of the basic Web server configuration that the next few sections will describe:



Note that it does not indicate machine configuration nor does it indicate what happens once a servlet receives a request. A later section, Designing a J2EE app with Domino and WSAS , will consider how servlets can interoperate with JSP pages, EJBs and the Domino server.

Starting the Domino server

First, make sure that the Lotus Domino Server has been started by making the appropriate selection from the Start menu, and then make sure that the Domino Administrator application has been started. Next, see if the LDAP, HTTP and the DIIOP servers are running. (DIIOP, the Domino Internet Inter-ORB Protocol, is necessary for a remote Java client to communicate with the Domino server.) To do this, type the following command in the server console:

```
>  show tasks
```

If they are running, the following lines should be in the listing that follows:

```
LDAP Server          Listen for connect requests on TCP Port:389
LDAP Server          Utility task
DIIOP Server         Listen for connect requests on TCP Port:63148
DIIOP Server         Utility task
HTTP Server          Listen for connect requests on TCP Port:80
DIIOP Server         Control task
```

It is likely that some are not running, in which case you'll need to add the following entries to the `ServerTasks` line in the `NOTES.INI` file in your Domino directory. You can use any text editor for this:

```
ServerTasks= <other entries...>
                                                ,HTTP,LDAP,DIIOP
```

Adding the Application Server DSAPI filter

Next, using the Domino Administrator, add the Application Server DSAPI filter. The Application Server DSAPI filter is a plug-in that allows Application Server to work with the Domino HTTP Server. To do this, you will need to have Application Server installed on the Domino Server machine -- even if you plan to run Application Server on another machine. Find the full path to the file `domino5_http.dll` in the Application Server installation. In Domino Administrator, select the current server document, select the Internet Protocols page and enter the full path and file name for the DSAPI filter:

With this DSAPI plug-in, HTTP requests that cannot be serviced by Domino will be forwarded to Application Server. This will be tested once additional configuration has been performed in Application Server.

To have these changes take effect, the easiest thing to do at this point is to restart the Domino server with the following command in the Domino server console:

```
    restart server
```

Verifying the Domino Web server

To verify that the Domino Web server is running, open a browser and enter the URL: `http://localhost`. This should bring up the Domino Web server start page:

Section 4. Configuring Application Server

Adding ncso.jar to the Application Server classpath

In order to use the Domino classes in an application running in Application Server, it is necessary to add the `.JAR` file containing them to the Application Server's classpath. To do this, locate the `ncso.jar` file in the Domino directory. It should be in the following directory if Domino is installed in `C:\Lotus\Domino` :

```
c:\Lotus\Domino\Data\Domino\Java\ncso.jar
```

Copy it to the Application Server's `lib\ext` directory, assuming Application Server

is in `C:\WebSphere\AppServer`, this would be:

```
c:\WebSphere\AppServer\lib\ext
```

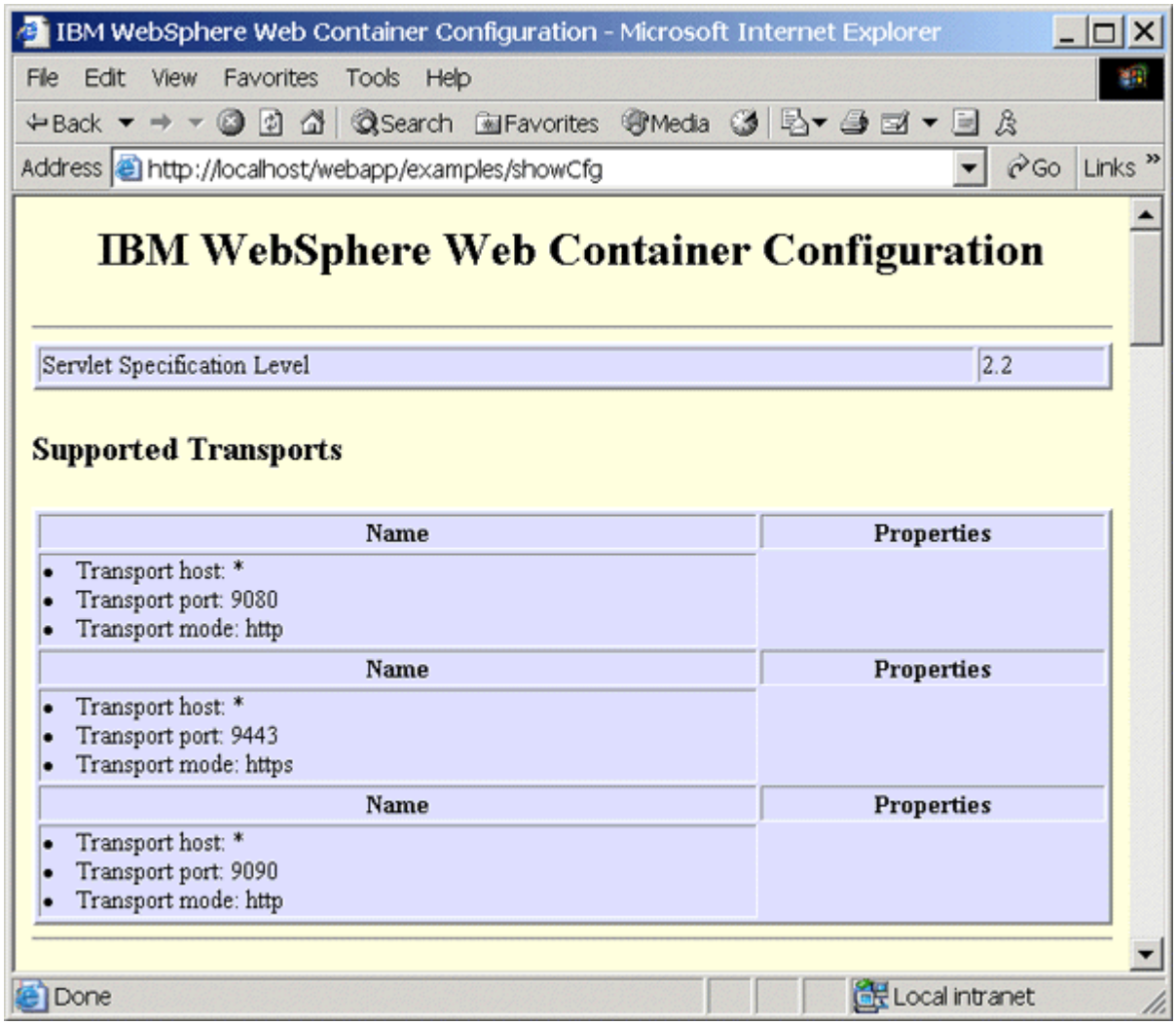
Application Server will search this directory, and the `.JAR` files in it, for classes.

Starting the WebSphere Application Server

Start the WebSphere Admin server. Depending on the version of Application Server, this may be a done from the Start menu by selecting **IBM WebSphere=> Application Server=> Start Admin Server**, or by starting it as a service using the service manager, for example. Again, this may vary according to the version of Application Server you are using; refer to the `readme.html` file in the directory where you installed the product for more information.

Verify that servlet requests are being forwarded and handled by Application Server by entering the following URL for showCfg, the sample servlet included with Application Server:

```
http://localhost/webapp/examples/showCfg:
```



Section 5. Designing a J2EE app with Domino and WSAS

J2EE design overview

This tutorial uses a sample application to demonstrate the key points of designing and building a J2EE application. Most issues are common to all Web applications, in particular, the desire to separate the presentation logic -- HTML, client-side scripting -- from the business logic. The next few panels, beginning with The Model-View-Controller design pattern, consider how the MVC approach can be implemented with Java servlets, JSP pages and EJBs.

Web applications commonly use relational databases, such as DB2 and Oracle, to

store all persistent data. However, some data, such as the messages in a newsgroup, are hierarchical in nature, and a relational database is not particularly well suited for this use. It is possible to do and is often done, but it isn't pretty. On the other hand, designing a hierarchical document store in Domino is simple. The sample application uses Domino to support this feature, and the sections that follow consider how to best incorporate it into the MVC design pattern.

The Model-View-Controller design pattern

Design patterns can most succinctly be described as condensed wisdom. They summarize the solution to a specific problem in a specific context based on experience. Patterns come in all shapes and sizes, but when discussing object-oriented programming, patterns usually refer to the relationship between a group of classes. For example, the Factory pattern is commonly used throughout the Java API to allow using abstract classes to instantiate concrete objects. Depending on its size and complexity, an application usually consists of at least a few patterns.

The MVC pattern is at a higher level of abstraction than class relationships. It describes the overall application design and is sometimes called the MVC framework or architecture. The MVC pattern neatly accomplishes the goal of separating presentation from business logic by dividing an application into three components:

- Model -- The data and its associated business logic
- View -- The presentation of the data
- Controller -- Means of manipulating the model via the view (to allow user interaction)

When these components are faithfully implemented, responsibilities are strictly divided among them. For example, the Model will likely contain dates, numbers and strings, but it should not have any role in formatting them. That is the sole job of the View.

Benefits of MVC

By encapsulating functionality in well-defined components with limited dependencies on other components, the MVC design provides many benefits, such as:

- Different teams, with different strengths, can work on different parts, largely independently. One team, comprising Java developers, may work on the Controller and Model, while another team, comprising Web designers, may work on the View.
- Components can be replaced. For example, a Web-based View could be replaced by a dedicated client application or by a Web services interface, with little or no change to the Model or Controller components.

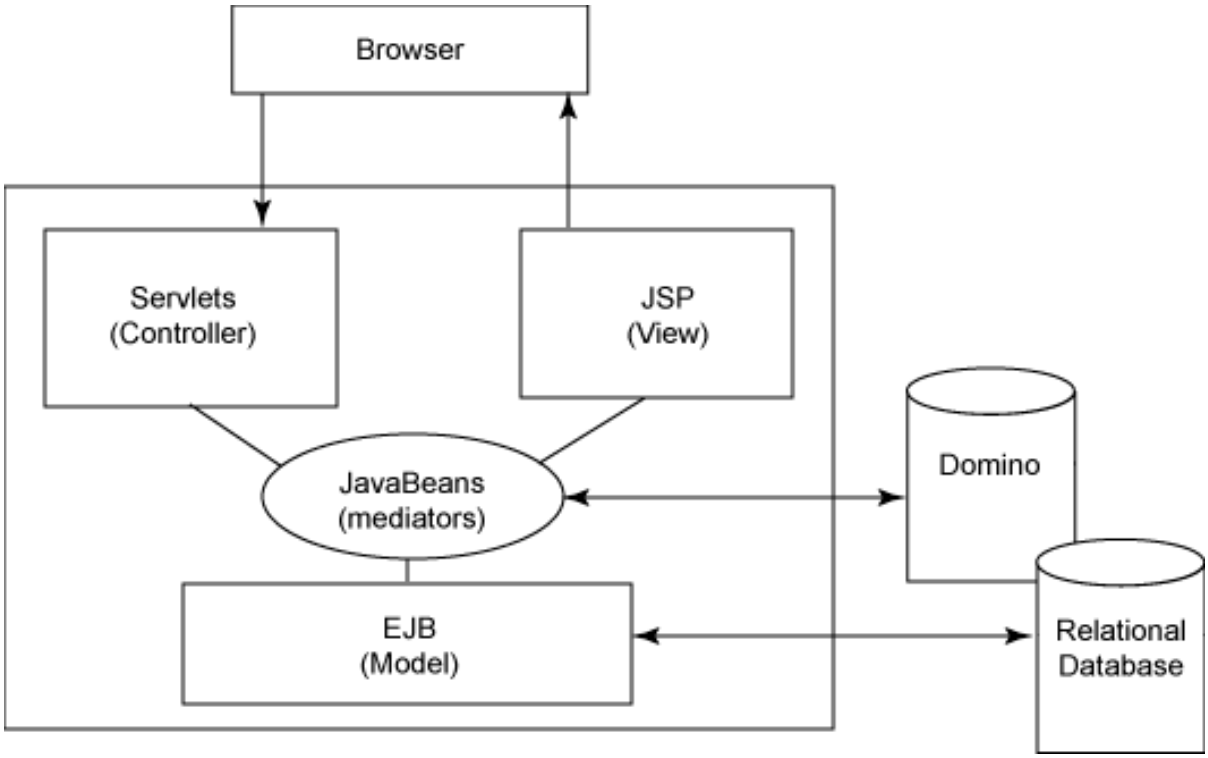
- Components can be deployed more flexibly. If performance becomes a problem because of increased traffic, for example, components can be moved onto separate servers.

Implementing MVC using J2EE

The MVC design is often implemented by using J2EE technologies as follows:

- View: JSP
- Controller: Servlets
- Model: EJB

In addition, JavaBeans are often used to mediate between each of these components. The following diagram illustrates a typical implementation of MVC that uses a standard database to provide persistence for EJBs:



Domino's place in MVC

Since each component in the J2EE implementation of the MVC design is written in the Java programming language (or, in the case of JSP pages, can contain Java code in the form of scriptlets), each can potentially access a Domino database.

There are two ways that Java programs, and potentially EJBs, can access Domino data. The first is to use the standard Java database technology, JDBC. Domino

provides a JDBC driver and can appear to be a standard relational database. But despite some SQL extensions that allow access to its hierarchical data, this limits the benefits that using Domino can provide. A much better way to access Domino data is to use the Domino classes for Java because these provide full access to the Domino object model.

With regard to EJB, no containers currently support using the Domino JDBC driver for container-managed persistence so it is only possible to use Domino with either session beans or with entity beans using bean-managed persistence. Provided they meet the application's requirements, session beans are perhaps the better choice since they are easier to use. In any case, EJB introduces complexity and overhead, and these factors need to be carefully weighed against the benefits they provide, particularly when considering the application's design and features.

In the sample application in this tutorial, Domino is accessed from a regular Java class, a controller class that populates a JavaBean with data.

Section 6. A sample J2EE application with Domino

The sample application: A discussion board

The sample application will be familiar to users of Lotus Notes and participants in Usenet groups. It is a discussion board that allows participants to post messages and reply to messages. In this section we'll demonstrate:

- Creating a discussion database using Notes.
- Writing a servlet that accepts browser requests and delegates them to controller classes. (Only one will be implemented here.)
- Implementing a controller class for handling requests for data from Domino. It uses data obtained from Domino to populate a JavaBean. It then forwards control to a JSP page for display.
- A JSP page that formats the data as HTML for presentation by the browser.

Key pieces of code will be presented in this section, but some, particularly exception handling, will be omitted. See Tools and configuration recommendations for a link to download the complete application.

A Domino discussion board database

Using the Notes client, we'll open a new discussion database based on the existing

discussion template. (If you prefer to customize the discussion or design something a little fancier, you can use Domino Designer to modify the template or create your own template.) Below is a step-by-step outline of the process:

1. Start Notes.
2. Select **File=> Database=> New**.
3. Select server.
4. Enter a name for the database; this example uses `WholeLottaTalking`.
5. Verify that the file name is acceptable.
6. Select template; this example uses the Discussion -- Notes and Web from Release 6.

Now, open the database:

1. Select **File=> Database=> Open**.
2. Select server.
3. Select database.

Enter a few sample documents and responses.

Creating the Web application

To begin creating the Web application using Application Developer, start Application Developer from the Start menu by selecting **Programs=> IBM Websphere Studio Application Developer**. (This may vary depending on where Application Developer was installed.) Perform the following steps in Application Developer:

- From the Perspective menu select **Open=> J2EE**.
- From the File menu select **New=> Enterprise Application Project**.
- Enter the application name; this example uses `DominoExampleEnt`.
- Uncheck Application client project name and EJB project name. These are not needed for this example.
- Change the Web project name to `DominoExample`. Leave the default location as suggested.
- Click **Finish**.

The project, `DominoExampleEnt`, will now appear under Enterprise Applications in

the J2EE view on the left side of the Application Developer window. The next step is to import the Notes classes. Press the Navigator tab at the bottom of the J2EE view. Right click on **DominoExample** and select **Properties** from the pop-up menu.

- On the left side of the panel click **Java Build Path**.
- On the right side of the panel, click the Libraries tab.
- Click **Add External JARs**.
- Use the file dialog box that appears to find and add the full path to the Domino `notes.jar` file. Assuming Domino is installed in `C:\Lotus\Domino`, this will be `C:\Lotus\Domino\notes.jar`.
- Click **OK**.

Creating the Java package

To create a package for the project, click on the plus sign next to `DominoExample` to expand the entries below it. To create a package for the source code:

- Right click on the Sources folder.
- From the pop-up menu, Select **New=> Other**.
- On the left side of the panel, select **Java**; on the right side select **Java package**.
- Click **Next**.
- Enter a name for Package, such as `org.gallardo.domino.example` and click **Finish**.

This last step will create a hierarchy of folders corresponding to the package name, the innermost being `example`.

Creating a Java class source file

These are the general steps required for creating a Java class source file. You can follow along and create the Java class file `DiscussionTree.java`, which will be used later:

- Right click on the innermost folder: `example`.
- Select **New=> Other** from the pop-up menu.
- To create a regular Java class: on the left side of the panel select **Java**; on the right side select **Java Class**.
- To create a servlet: on the left side of the panel select **Web**; on the right side select **Class**.

- Click **Next**.
- Enter the class name, for example, `DiscussionTree`.
- Change access modifiers, if appropriate.
- Click **Finish**.

This will automatically produce code like the following (in this case, for the regular class with the default superclass `java.lang.Object` which, of course, does not need to be specified):

```
package org.gallardo.domino.example;
public class DiscussionTree {
}
```

Creating the servlet class

Create the servlet class, `DominoExampleServlet`, by following the directions in the previous panel. It's important not to create this as a standard Java class but as a servlet, because in addition to creating the class, the servlet wizard creates essential deployment information.

Add the following imports -- after the package statement -- to the source that is generated automatically:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*
```

To process a request in a servlet, one or both of the following methods should be implemented:

- `doPost(HttpServletRequest, HttpServletResponse)`
- `doGet(HttpServletRequest, HttpServletResponse)`

The skeletons of these functions will be generated automatically when you create the servlet class. For consistency in implementing `doGet()` and `doPost()`, add to each a call to the generic `processRequest()` method.

Next, provide the code for the `processRequest()` method (perhaps by cutting and pasting from the source files provided in the sample application ZIP file).

Controller helper classes

The method `processRequest()` is called by `doGet()` and `doPost()` so that all requests can be handled by a single method. A servlet would typically process these

requests based on the request parameters and the session's attributes. There are a number of different approaches to doing this. One way is to have different servlets that handle different requests.

A better way is to have a single servlet handle requests initially and then delegate to other classes. In this example, the servlet calls an abstract `HttpController` class; the `HttpController` class has a static factory method that returns an appropriate concrete controller class based on a REQUESTPAGE `request` parameter:

```
        HttpController ctrl = HttpController.createController(
                req.getParameter("REQUESTPAGE"));
        ctrl.process(req, resp, getServletContext);
}
```

The `process()` method needs the request and response parameters because it will need them in to order to forward them to the JSP page once it has obtained the data and put it in a JavaBean. It also needs a `ServletContext` to forward the request, but it can't obtain the servlet context because the controller helper class is not a servlet, so this also needs to be passed in as a parameter.

Processing the request

Create the `HttpController` class as a regular Java class. Change the default access modifiers to include abstract. Add the following imports -- after the package statement -- to the code that is generated:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

The abstract `HttpController` class has one concrete method, a static method that instantiates a concrete controller class based on a `String` parameter. It should use a properties file (or some other configurable means) to map the String literal representing the type to the proper concrete class. In this example only one controller class is implemented, so the value is hard-coded:

```
public static HttpController createHTTPController(String type) {
        return new StartPageController();
}
```

The concrete implementation, `StartPageController`, extends the `HttpController` class and implements the `process()` method. This is the class that does the actual work of obtaining the data from the Notes database, instantiating a JavaBean class to hold the data and then forwarding the request to a JSP page that will format and return the data to the browser as HTML.

Connecting to a Notes database

The code for obtaining data from Notes will be in a concrete subclass of `HttpController`. Create a class `StartPageController`. Enter `HttpController` as the superclass but leave the abstract modifier unchecked (since of course, this is a concrete implementation of the abstract `HttpController` class). Add the following imports to the class code that is generated:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.domino.*;
```

The `StartPageController` needs to:

- Obtain a Notes session.
- Open the database.
- Open a view.
- Obtain a ViewNavigator.

Here is the code for these first steps:

```
public void process(
    HttpServletRequest req,
    HttpServletResponse resp,
    ServletContext servctx)
    throws ServletException, IOException {

    Session nses = null;
    try {
        String server = "noizmaker";
        String user = "dgallard";
        String pwd = "minfishy";
        String dbname = "wholelot.nsf";
        nses = NotesFactory.createSession(server, user, pwd);
        Database db = nses.getDatabase("", dbname);

        if (!db.isOpen()) {
             db.open();
        }

        String viewname = "All Documents";
        View view = db.getView(viewname);
        ViewNavigator viewnav = view.createViewNav();
        ViewEntry ventry = viewnav.getFirst();
```

Reading data from a Notes database

Once a `ViewNavigator` has been obtained, it can be used to iterate through the database view. By calling the `getFirst()` and `getNext()` methods, each document, category, and possibly total will be returned in the order they appear in

the view. This example is only concerned with obtaining a top-level view of the documents, so categories and totals will be ignored.

Two pieces of information will be saved for each document: its topic and its indentation level. They'll each be stored in vectors:

```
Vector text = new Vector();
Vector indent = new Vector();
while (ventry != null) {
         if (ventry.isDocument()) {
               int in = ventry.getIndentLevel();
               indent.add(new Integer(in));
               Vector colvals = ventry.getColumnValues();
               text.add(colvals.elementAt(3));
               String val = colvals.elementAt(3).toString();
               ventry = viewnav.getNext();
        }
}
```

The Model: JavaBean

The MVC design suggests that the data be stored in a way that is independent of any presentation issues. JavaBeans are a good way to do this because they provide a convention for setting and getting data, and JSP technology is able to work well with them.

Representing a hierarchical tree is a bit of a challenge, however, because in order for the JSP page to use JavaBeans automatically, the `get` methods should not take any parameters. This requires a bit of a compromise. In this implementation, because multiple rows will be stored in a single JavaBean, vectors will be used internally and the `get` methods will take a parameter representing the index of the requested value. This means that the JSP page will need to process it by looping and explicitly calling methods in the JavaBean to retrieve each row's values.

Create the JavaBean class, `DiscussionTree`, the same way as the previous classes, with `java.lang.Object` as the superclass. Below is the completed class:

```
package org.gallardo.domino.example;

import java.util.*;

public class DiscussionTree {
    private Vector indent = null;
    private Vector text = null;

    public DiscussionTree()
    {
    }

    public DiscussionTree(Vector i, Vector t)
    {
        indent = i;
        text = t;
    }

    public int getSize() {
        if (text.size() == indent.size()) {
            return text.size();
        } else {
```

```
            return 0;
        }
    }

    public String textAt(int i) {
        return (String) text.elementAt(i);
    }

    public int indentAt(int i) {
        return ((Integer)indent.elementAt(i)).intValue();
    }
}
```

Populating the JavaBean

The controller class instantiates the `DiscussionTree` JavaBean by calling the constructor with the two vectors containing the data from the view, and sets the JavaBean as an attribute associated with the request:

```
        DiscussionTree dt = new DiscussionTree(indent, text);
        req.setAttribute("DiscussionTree", dt);
```

A JavaBean can be associated with one of four scopes: application, session, page, or request. Briefly, application scope means that the JavaBean will be available to any other servlet in the Web application. Session scope means that the JavaBean will be available to the current session as long as it's active. Page and request scope are similar and mean that the JavaBean will be available for the duration of the current request. Calling the `setAttribute()` method of the request parameter that was obtained from the servlet associates it with the current request. Forwarding the request to a JSP page will make the JavaBean, in this case, an instance of the `DiscussionTree` class, available to it.

The following code forwards the request to a JSP page called `discussiontree.jsp` :

```
        RequestDispatcher rd =
                servctx.getRequestDispatcher("/discussiontree.jsp");
        rd.forward(req, resp);
```

The View: JSP

The JSP page is responsible for displaying the data. It should consist of HTML and as little code as possible. Locate the `webApplication` directory under the `DominoExample` project -- this is the directory for HTML, JSP and other resource files. To create the JSP file:

- Right click on the `webApplication` folder.
- Select **New=> Other** from the pop-up menu.
- On the left side of the panel select **Web**; on the right side select **JSP File**.

- Click **Next**.
- Enter `discussiontree.jsp` as the name.

For this example, because it's necessary to iterate through the set of rows that were returned from Domino, there will be some Java code in the form of scriptlets. First, a `jsp:useBean` tag is used to obtain the JavaBean created in the servlet. It takes three parameters: an identifier (which must correspond to the name used in the `setAttribute` method in the servlet), the class name, and the scope:

```
<jsp:useBean id="DiscussionTree"
             class="org.gallardo.domino.example.DiscussionTree"
                scope="request" />
To access the data, Java code can be embedded directly in a JSP page by prefacing
   it with <% and closing with >.
Note how it can be broken up and intermingled with HTML:
<%
  for(int i= 0; i<DiscussionTree.getSize() ;i++)
  {
      switch(DiscussionTree.indentAt(i))
      {
        case 0:  %> <IMG SRC="root.gif"> <% ;
                    break;
              case 1:  %> <IMG SRC="indent_1.gif"> <%
                    break;
          case 2:  %> <IMG SRC="indent_2.gif"> <%
                    break;
          default: %> <IMG SRC="indent_3.gif"> <%
      }
%>
    <%= DiscussionTree.textAt(i) %><P>
<% } %>
```

Custom tag libraries

An alternative to using scriptlets, (or even servlets), is to use JSP custom tags. Custom tags, introduced in JSP 1.1, allow a developer to create a class in Java that can be called from a JSP page using custom tags. These tags are grouped into libraries using an XML file called a *tag library descriptor file*, with an extension `.tld`, that associates the classes with tag names that can be used in JSP pages. Using a custom tag is similar to using JavaBean except that much more complicated behavior can be implemented in custom tag class.

Domino 6.0 includes a custom tag library that greatly extends what can be done with JSP pages and Domino. This provides an easy way to access Domino forms and data entirely within JSP pages. Depending on application requirements, this can be a convenient alternative to using servlets and the Domino Java classes directly.

Custom tags are often a good way to minimize the amount of Java code in JSP pages. In the case of the current example, however, the individual elements need to be formatted. This formatting is a job for the JSP page -- little would be gained from writing custom tags since the looping behavior needs to be implemented in the JSP page regardless.

See the Tutorial resources section for references to two sample JSP applications available from the Lotus Developer Domain -- a Discussion application and Document Review and Feedbac application -- that demonstrate the use of the Domino custom tag library to build complete applications with JSP technology.

Importing additional resources

The `discussiontree.jsp` requires some graphic elements. These can be created in Application Developer, but for this example they'll be imported. They are included with the source code for this application. After unzipping the sources, import the `.gif` files as follows:

- Select **webApplication** by clicking on it once.
- From the File menu, select **Import**.
- In the panel that appears, select **File system** and click **Next**.
- Click the Browse button next to the Directory field (not the Folder field) to locate the directory where the `.gif` files are located and click **OK**.
- On left side of the panel that appears, click on the directory name.
- On the right side of the panel that appears, check each of the `.gif` files.
- Leave the destination as is.
- Click **Finish**.

Section 7. Deploying the Web application

Creating a WAR: the Web application file

Web applications are a set of resources -- Java classes, JSP files, HTML and images -- that work together to present a set of related Web pages with which a user can interact through a browser. The standard format for an Web application is a `.WAR` file, which is essentially an archive file like a `.JAR` file, but with a well-defined, standard directory structure and a number of XML files describing the details of how it is to be deployed. To create the `.WAR` file for the DominoExample application:

- Right click on the Web project name, `DominoExample`.
- From the pop-up menu, select **Export WAR**.
- In the pane that appears, ensure that is says DominoExample under "What resources do you want to export?".

- Click on **Browse** to select a directory for the `.WAR` file, such as the `InstallableApps` in the Application Server directory. This may be, for example, `C:\WebSphere\AppServer\InstallableApps`.
- Enter a name for the `.WAR` file such as `DominoExample.war`.
- Click **Finish**.

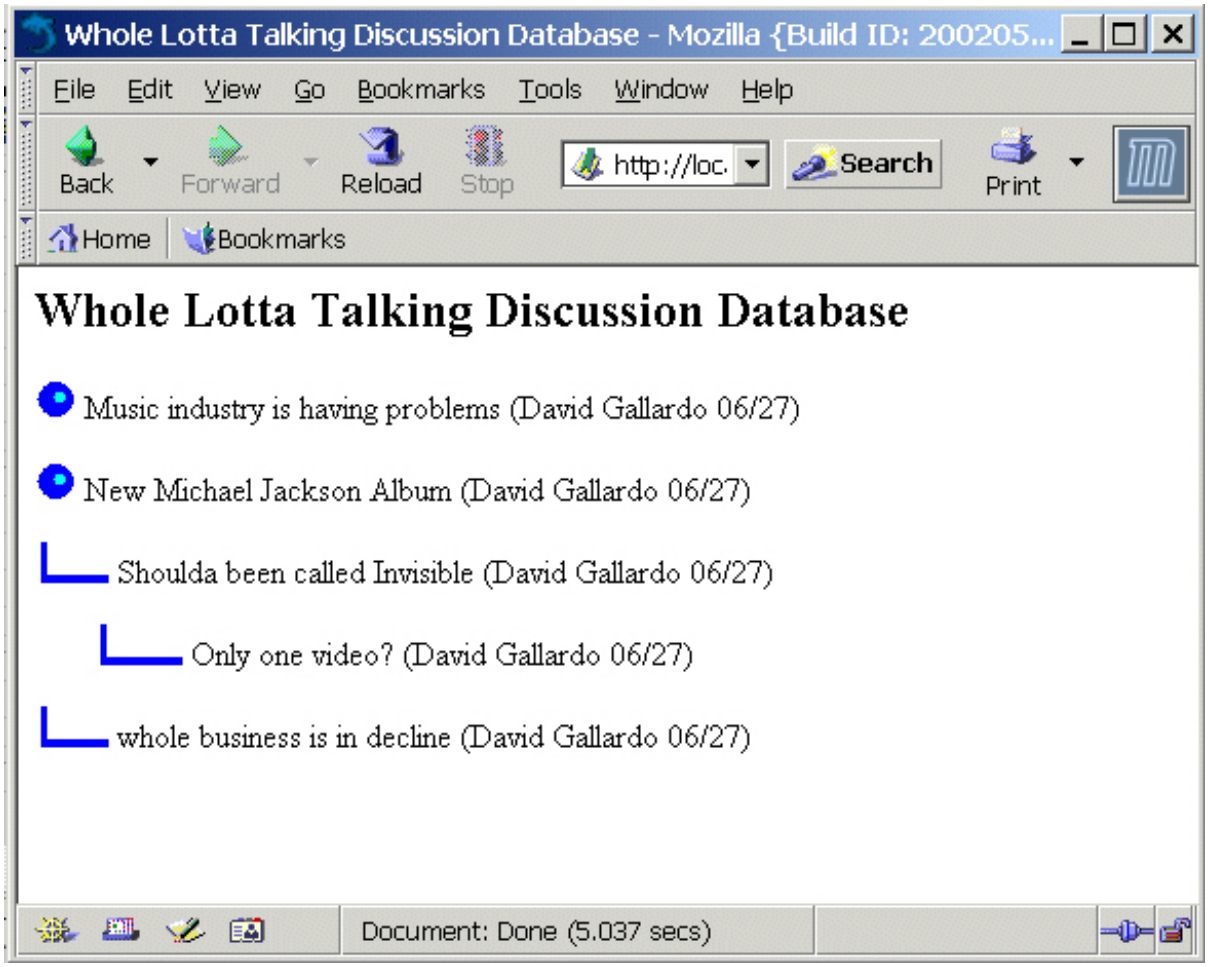
Deploying the enterprise application in Application Server

To deploy the Web application, the Application Server should be running. Start the Administrator's Console from the Start menu by selecting **IBM WebSphere=> Application Server=> Start Admin Console**. After logging in, click on the plus sign for nodes to expand the tree, then click on the plus sign for your node. To install the application:

- Click on **Enterprise Application**. The panel on the right will show the applications that are currently installed.
- Click the Install button.
- In the Application Installation Wizard that appears, click **Browse** to find the `.WAR` file created in Application Developer, `DominoExample.war`.
- Enter a name for the application, such as `DominoExample` and a context root -- that is, the part of the URL that follows the server name -- for example, `DominoWebExample`.
- Click **Next**.
- Accept the defaults in the next screen and click **Next**.
- Accept the defaults in the next screen and click **Finish**. You will be returned to the Enterprise Applications screen. The DominoExampleWeb application will now appear in list of applications on this screen.
- Near the top of the screen will be a message that the Plug configuration needs to be regenerated. Click on this message.
- On the next page, click the Generate button.
- Check the box next to it and click **Start** to start it.

Running the Web application

To run the Web application, start a browser and type `http://localhost/DominoWebExample/DominoExampleServlet`. This should bring up a screen like the following:



Next steps

If an error occurs when you try to run the application, you may wish to verify the properties of the Web Application. In particular, you will want to make sure that it started. It may be necessary to shut down the Application Server and restart it. To do this, go to a command prompt and change to the Application Server's `/bin` directory. This may be, for example, `C:\WebSphere4.0\AppServer\bin`. Enter:

```
stopserver
```

Wait for the Application Server to shutdown and then enter:

```
startserver
```

If you have trouble using the Internet shortname and password, you may wish to change the Internet security settings. In the Domino Administrator, go to the server's Security page and, under the section "Internet Access," change Internet Authentication to "More name variations with lower security."

If all else fails, refer to the Application Server online documentation by clicking **Help** on the Administrator's Console screen.

Once the application is working, you may wish to consider how it could be extended to be more useful. One idea is for the JSP page to dynamically include buttons to open the document. To support that, the `DiscussionTree` bean would need to include a way of identifying documents so that the JSP page could pass the ID back to the servlet in an attribute, and set REQUESTPAGE to point to a new controller that retrieves an individual document from Domino.

This sample application only sketches out some of the things that can be accomplished using Domino in a J2EE environment. The key point is that it demonstrates how Domino can be used to add collaborative features to an application quickly. Consider the amount of work that would be required to implement a discussion feature from scratch using a relational database.

Section 8. Summary

Tutorial summary

This tutorial covered:

- Key strengths of Lotus Domino and WebSphere Application Server and how they can best be used together.
- How Domino can be used with Application Server to build a J2EE application.
- Details of configuring Domino and Application Server to work together.
- An introduction to designing applications using J2EE and the Model-View-Controller design.
- A sample application that uses a servlet, a JavaBean and a JSP page to implement a J2EE application that displays information from a Domino discussion database.

Resources

Learn

- For a Redbook that covers Domino and WebSphere in detail and includes a large sample application that integrates Domino, servlets, JSP pages and EJBs, see Domino and WebSphere Together Second Edition.
- For a Redbook covering Runtime patterns for Domino and WebSphere integration, see Applying the Patterns for e-business to Domino and WebSpere Scenarios.
- For a Redbook covering integrating Domino, Java, WebSphere and other technologies from a variety of different angles, see Connecting Domino to the Enterprise using Java.
- See the tutorial, Building dynamic Web sites with WebSphere Studio, for information on using WebSphere Application Developer 4.0 (*developerWorks*, March 2002)
- For novices seeking information about JSP pages, see Introduction to Java Server Pages. (*developerWorks*, August 2001)
- For a short tutorial covering the basics of servlets, see Building Java HTTP servlets. (*developerWorks*, September 2000)
- For a kit that contains useful documentation about using Domino with Java, including complete documentation for the Domino Java classes, download the : Lotus Domino Toolkit for Java/CORBA (5 MB).
- For comprehensive coverage of using Visual Age for Java to develop J2EE for WebSphere Application Server and some solid design advice, see *Enterprise Java Programming with IBM WebSphere*, by Kyle Brown (Editor), Gary Craig, Greg Hester, Jaime Niswonger, David Pitt, and Russell Stinehour (Addison-Wesley, 2001).
- *Core J2EE Patterns: Best Practices and Design Strategies*, by Deepak Alur, John Crupi, and Dan Malks (Sun Microsystems Press, 2001) is the bible on design patterns for J2EE.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- The following sample applications demonstrating the use of the Domino tag libraries for JSP are available for free from the Lotus Developer Domain:
  - Access the Document Review and Feedback Application.
  - Access the Discussion Application.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.

About the author

David Gallardo
David Gallardo is an independent software consultant and author specializing in software internationalization, Java Web applications and database development. He has been a professional software engineer for over 15 years and has experience with many operating systems, programming languages and network protocols. His recent experience includes leading database and internationalization development at a business-to-business e-commerce company, TradeAccess, Inc. Prior to that, he was a senior engineer in the International Product Development group at Lotus Development Corporation where he contributed to the development of a cross-platform library providing Unicode and international language support for Lotus products including Domino.