ORACLE

Mastering J2EE Application Development Series

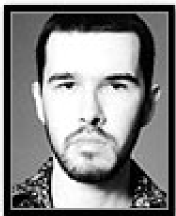
Step 4 of 12: Building on Frameworks

Frameworks To The Rescue

by Andrei Cioroianu

Build complex Web UIs using Oracle's standard JSF components — ADF Faces.

Downloads for this article
- Andrei Cioroianu's Code Samples
- ADF Faces
- Oracle JDeveloper 10*g*



Many standard J2EE technologies and the APIs they expose are not as easy-to-use as framework-based approaches. (J2EE's complexity is the price we've had to pay for its flexibility and extensibility — without which we wouldn't have the number of good Java frameworks in existence today.) Nevertheless, even as J2EE standards evolve, some areas (such as UI) still need improvement, and it's in areas like these that frameworks can be especially useful to developers.

For example, Servlets, JSP, (JavaServer Pages) and JSF (JavaServer Faces) do not support file uploading, a capability required in many different types of applications. However, these J2EE technologies do allow frameworks to plug in custom implementations of many features, including a file-upload capability.

In addition to implementing missing features, J2EE frameworks, such as Oracle Application Development Framework (ADF), provide the infrastructure for integrating various enterprise technologies; help you to exploit design patterns such as MVC; and allow IDEs like Oracle JDeveloper to generate source code that is compact and maintainable.

Most importantly, by building your applications on frameworks, you become more productive because you spend less time coding, debugging, and deciding what patterns to use, how to implement a particular feature, or how to deal with some technical problem that was already solved by others.

"Many standard J2EE technologies and the APIs they expose are not as easy-to-use as framework-based approaches."

This article shows you how to build a Web application that lets users upload files. The key technologies used are JSF and ADF Faces. You'll learn how to configure the application to use the JSF core component libraries together with the ADF Faces component set. The examples in this article were tested with Oracle Application Server Containers for J2EE (OC4J)10*g* (9.0.4), JSF 1.1.01, and ADF Faces EA10. (Since the ADF Faces version used was in an Early Access stage, you'll have to change the code if you want to run the examples with a different ADF Faces version than the version used in this article. In this case, make sure that you change EA10 from the URLs of the tag libraries used in the JSF pages.)

Choosing Frameworks

Before building our example application on JSF and ADF Faces, let's see how we end up choosing them. Deciding which

frameworks to use isn't easy because there are so many Java frameworks covering all aspects of the application development process. In this article we'll focus on UI frameworks since we want to build a Web application that uploads files. Developers choose frameworks depending on their current skills, experience, the reviews and articles they read, the opinions of their friends and colleagues, but it's a good idea to evaluate the frameworks yourself taking into account factors such as the support for standards, license, functionality, maturity, and extensibility. When choosing a new framework or when switching to another framework, you should ask at least the following set of questions.

Questions	Remarks	UI Frameworks Pros and Cons
Is the framework industry standard? What standards does the framework support? Are there important standards that aren't supported?	Most Java Web frameworks support Servlets and/or JSP. JSF is a standard too, but it's not so widely supported since many Java frameworks were developed before JSF.	There are attempts to integrate Struts and JSF, but Struts doesn't implement the JSF standard. ADF Faces provides a set of UI components that work well with any implementation of the JSF standard.
Is the framework open source? Does the license allow me to distribute my applications packed together with the framework? Do I have access to the source code? May I modify it? May I distribute the modified version?	The access to the source code may prove to be very important in order to understand well the framework. Changes of the framework's code are rarely needed if the framework is well designed. However, having the legal rights to change the framework, if necessary, is a plus.	The Struts framework is distributed under the Apache License, which allows modification. JSF is a JCP standard and is distributed under a more restrictive license, but the source code is available.
Is the framework comprehensive? Does the framework provide everything I need? Is it easy to use? Can I use it together with other complementary frameworks?	Some frameworks are designed for developers who prefer to write code by hand. Other frameworks provide support for code-generation tools. A good framework, however, should allow developers to take either approach.	The JSF core component library provides almost the same functionality as the Struts' form tags. JSF has the advantage of allowing component providers to easily build new and extend existing components in order to add functionality. ADF Faces is such a component set built on top of JSF that adds many features, such as support for file uploading. ADF Faces offers rich UI components that have no equivalent in Struts and JSF.
Is the framework mature? Does it have a large user base? Are bugs fixed quickly? What kinds of projects have been built with the framework?	Many small frameworks do not evolve beyond their first release. A framework supported by a large developer community is a much safer bet.	Struts is more mature than JSF, but JSF is gaining acceptance.
Is the framework extensible? What are the mechanisms used to extend the framework? Will future versions support those mechanisms?	Extensibility is very important because it allows you to customize the framework for the needs of your application.	JSF is very extensible, enabling you to develop custom components, renderers, validators, data models, converters, and event listeners. ADF Faces is a set of custom components that takes advantage of JSF's extensibility. Struts is less extensible than JSF.
Is the framework widely supported? Are there IDEs supporting the framework? How many? What kind of support do they provide?	A good IDE can significantly increase your productivity. Any framework should be designed so that it can be easily supported by a variety of development tools.	Struts is already supported by most Java IDEs, including Oracle JDeveloper. Support for JSF is being added to many IDEs. A Developer Preview of JDeveloper is already available that supports JSF and custom JSF components such as ADF Faces.

The simple application presented in this article could be developed using either Struts or the JSF/ADF Faces combination. Both Struts and JSF provide a complete set of tags for building Web forms. However, each of them has pros and cons as described in the table above. The open source license and the large numbers of projects completed successfully makes Struts very appealing. On the other hand, JSF is a Java standard, so it is gaining support from the tool vendors every day. If you need rich UI components and a consistent look-and-feel, JSF in combination with ADF Faces components beats the competition.

Configuring JSF and ADF Faces

The Web application, the JSF framework, and ADF Faces are configured using the three XML files that are presented in this section (web.xml, faces-config.xml, and adf-faces-config.xml, respectively).

Before trying to work through this example, be sure to set up your development environment as detailed in "How To Use ADF Faces With JDeveloper 10*g*" (listed in the Resources section at the end of this article).

Creating the Web application descriptor. The web.xml file starts with an initialization parameter (`DATA_DIR`) whose value indicates where the application will store the uploaded files:

```
<web-app>

    <context-param>
        <param-name>DATA_DIR</param-name>
        <param-value>/WEB-INF/data</param-value>
    </context-param>
    ...
</web-app>
```

The `DATA_DIR` parameter is specific to the example application in this article. When developing your own applications, you are free to do anything you want with the uploaded files. For example, you could store their content in a database.

Our Web application configures the FacesServlet (in web.xml) like any other JSF-based application, mapping the controller servlet to the `*.faces` resources:

```
<web-app>
    ...
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>

    <servlet>
        <servlet-name>FacesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>FacesServlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

The AdfFacesFilter makes sure that the ADF Faces framework is properly initialized and provides support for the `multipart/form-data` encoding that must be used when uploading files. The ADF filter is mapped to the FacesServlet so that it can process any `*.faces` request:

```
<web-app>
    ...
    <filter>
        <filter-name>AdfFacesFilter</filter-name>
        <filter-class>oracle.adf.view.faces.webapp.AdfFacesFilter</filter-class>
        <init-param>
            <param-name>oracle.adf.view.faces.UPLOAD_MAX_MEMORY</param-name>
            <param-value>50000</param-value>
        </init-param>
        <init-param>
            <param-name>oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE</param-name>
            <param-value>1000000</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>AdfFacesFilter</filter-name>
        <servlet-name>FacesServlet</servlet-name>
    </filter-mapping>
    ...
</web-app>
```

The web.xml file configures a home page (index.jsp) for our application, as well as another JSP page (SizeError.jsp) that handles any java.io.EOFException thrown by the AdfFacesFilter when the user attempts to upload a file whose size exceeds the `UPLOAD_MAX_DISK_SPACE` limit configured above.

```
<web-app>
    ...
    <error-page>
        <exception-type>java.io.EOFException</exception-type>
        <location>/SizeError.jsp</location>
    </error-page>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

The SizeError.jsp page uses JSP Standard Tag Library (JSTL) to parse web.xml and to obtain the `UPLOAD_MAX_DISK_SPACE` parameter, whose value is reported to the user in an error message formatted with JSTL:

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

<c:if test="${empty maxDiskSpace}">
    <c:set var="maxDiskSpaceParam"
        value="oracle.adf.view.faces.UPLOAD_MAX_DISK_SPACE"/>
    <c:import var="xml" url="/WEB-INF/web.xml"/>
    <x:parse varDom="dom" xml="${xml}"/>
    <x:forEach var="filter" select="$dom/web-app/filter">
        <x:forEach var="filterParam" select="$filter/init-param">
            <x:if select="$filterParam/self::node()[param-name=$maxDiskSpaceParam]">
```

```
                <x:set var="maxDiskSpace" scope="application"
                    select="string($filterParam/param-value/text())"/>
            </x:if>
        </x:forEach>
    </x:forEach>
</c:if>

<fmt:setBundle var="resources" basename="adfupload.Resources"/>
<fmt:message bundle="${resources}" key="FILE_IS_TOO_LARGE">
    <fmt:param value="${maxDiskSpace}"/>
</fmt:message>
```

The index.jsp page forwards the HTTP request to the form page that enables the file uploading:

```
<jsp:forward page="UploadForm.faces"/>
```

Creating the JSF configuration file. The faces-config.xml file defines a managed bean (adfupload.UploadBean) whose source code is presented in the next section of this article. JSF will create instances of this bean in the JSP `request` scope. The `uploadedFile` and `override` properties will be initialized by JSF with `null` and `true`:

```
<faces-config>

    <managed-bean>
        <managed-bean-name>uploadBean</managed-bean-name>
        <managed-bean-class>adfupload.UploadBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>uploadedFile</property-name>
            <null-value/>
        </managed-property>
        <managed-property>
            <property-name>override</property-name>
            <value>true</value>
        </managed-property>
    </managed-bean>
    ...
</faces-config>
```

If the outcome of the UploadForm.jsp is OK, JSF forwards the HTTP request to UploadResult.jsp:

```
<faces-config>
    ...
    <navigation-rule>
        <from-view-id>/UploadForm.jsp</from-view-id>
        <navigation-case>
            <from-outcome>OK</from-outcome>
            <to-view-id>/UploadResult.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
    ...
</faces-config>
```

The `adfupload.Resources` bundle is configured as the message bundle of the application:

```
<faces-config>
    ...
    <application>
        <locale-config>
            <default-locale>en</default-locale>
        </locale-config>
        <message-bundle>adfupload.Resources</message-bundle>
    </application>

</faces-config>
```

Creating the ADF configuration file. ADF initialization parameters, such as `accessibility-mode` and `look-and-feel`, must be specified in a separate file named adf-faces-config.xml

```
<?xml version="1.0"?>
<adf-faces-config xmlns="http://xmlns.oracle.com/adf/view/faces/config">
    <debug-output>true</debug-output>
    <accessibility-mode>#{prefs.proxy.accessibilityMode}</accessibility-mode>
    <look-and-feel>#{prefs.proxy.lookAndFeel}</look-and-feel>
</adf-faces-config>
```

Using the APIs Provided by ADF Faces and JSF

The `UploadBean` class contains the Java code that implements the logic of the example Web application. When developing your own applications, you'll have to develop similar JavaBeans for handling the uploaded files, using the same JSF and ADF Faces APIs.

UploadedFile properties. As you'll see later in this article, the UploadForm.jsp page generates an <input type="file" ...> form element with the <af:inputFile> tag provided by ADF Faces. The information about the uploaded file is stored in a bean property whose type is `oracle.adf.view.faces.model.UploadedFile`. Note that `UploadedFile` is an interface but you don't have to implement it. When the user selects a file and submits the form, ADF Faces gets the content of the uploaded file, creates an instance of UploadedFile, and sets your bean property. Then, you may call its methods in order to obtain the filename, the file size, and the content type, and you can read the uploaded content from an input stream returned by the `getInputStream()` method of `UploadedFile`. In addition to one or more `UploadedFile` properties, your JavaBeans may have, of course, other properties specific to your applications, such as the `override` property of our `UploadBean` example:

```
package adfupload;

import oracle.adf.view.faces.model.UploadedFile;

public class UploadBean {
    ...
    private UploadedFile uploadedFile;
    private boolean override;

    public void setUploadedFile(UploadedFile uploadedFile) {
        this.uploadedFile = uploadedFile;
    }

    public UploadedFile getUploadedFile() {
        return uploadedFile;
    }

    public void setOverride(boolean override) {
```

```
        this.override = override;
    }

    public boolean getOverride() {
        return override;
    }
    ...
}
```

After extracting all the information you need from the `UploadedFile` instance, you should call its `dispose()` method in order to free all resources allocated for the uploaded file. After this method has been called, all properties of the `UploadedFile` instance lose their values. Therefore, our `UploadBean` stores the filename, the file size, and the content type into private fields, making their values available as read-only properties:

```
public class UploadBean {
    ...
    private String fileName;
    private long fileSize;
    private String contentType;

    public String getFileName() {
        return fileName;
    }

    public long getFileSize() {
        return fileSize;
    }

    public String getContentType() {
        return contentType;
    }
    ...
}
```

Adding error messages. The `error()` method of `UploadBean` adds an error message to the current `FacesContext` and throws a `ServletException`:

```
import javax.faces.context.FacesContext;
import javax.faces.application.FacesMessage;
import javax.servlet.ServletException;
import java.util.ResourceBundle;

public class UploadBean {
    ...
    private void error(String compId, String messageId,
            FacesMessage.Severity severity) throws ServletException {
        FacesContext context
            = FacesContext.getCurrentInstance();
        ResourceBundle bundle = ResourceBundle.getBundle(
            context.getApplication().getMessageBundle());
        FacesMessage message = new FacesMessage(
            bundle.getString(messageId));
        message.setSeverity(severity);
        context.addMessage(compId, message);
        throw new ServletException(message.getSummary());
```

```
    }
    ...
}
```

The JSF and ADF components report error messages to the user. These messages are loaded from a resource bundle, such as our `Resources.properties`, which may also keep the labels of the UI components:

```
title=JSF File Uploading with ADF Faces
file=File
override=Override
upload=Upload
fileName=File Name:
fileSize=File Size:
contentType=Content Type:
backLink=Back to UploadForm

MISSING_DATA_DIR_PARAM=Missing DATA_DIR parameter
CANNOT_ACCESS_DATA_DIR=The data directory is not accessible
CANNOT_OVERRIDE=The file already exists
IO_EXCEPTION=An error occurred while saving the file
FILE_IS_TOO_LARGE=Uploaded files may not have more than {0} bytes
```

Using initialization parameters. In order to obtain the values of the application parameters from web.xml, our UploadBean must get the ServletContext:

```
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.servlet.ServletContext;

public class UploadBean {
    ...
    private ServletContext getServletContext() {
        FacesContext facesContext
            = FacesContext.getCurrentInstance();
        ExternalContext externalContext
            = facesContext.getExternalContext();
        ServletContext servletContext
            = (ServletContext) externalContext.getContext();
        return servletContext;
    }
    ...
}
```

The `DATA_DIR` parameter indicates the directory where the uploaded files should be saved. The value of this parameter is converted to a system path with the `getRealPath()` method of ServletContext.

```
import javax.faces.application.FacesMessage;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import java.io.File;

public class UploadBean {
    public static final String FILE_COMP_ID = "file";
    public static final String MISSING_DATA_DIR_PARAM_ERROR
        = "MISSING_DATA_DIR_PARAM";
```

```
    public static final String CANNOT_ACCESS_DATA_DIR_ERROR
        = "CANNOT_ACCESS_DATA_DIR";
    public static final String DATA_DIR_PARAM
        = "DATA_DIR";
    ...
    private File getDataDir() throws ServletException {
        ServletContext application = getServletContext();
        String dataDir = application.getInitParameter(DATA_DIR_PARAM);
        if (dataDir == null || dataDir.length() == 0)
            error(FILE_COMP_ID, MISSING_DATA_DIR_PARAM_ERROR,
                FacesMessage.SEVERITY_FATAL);
        String realDir = application.getRealPath(dataDir);
        if (realDir == null)
            error(FILE_COMP_ID, CANNOT_ACCESS_DATA_DIR_ERROR,
                FacesMessage.SEVERITY_FATAL);
        return new File(realDir);
    }
    ...
}
```

Saving the uploaded file. The `saveUploadedFile()` method saves the content of the uploaded file into a given target file:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;

public class UploadBean {
    ...
    private void saveUploadedFile(File targetFile) throws IOException {
        InputStream in = new BufferedInputStream(
            uploadedFile.getInputStream());
        try {
            OutputStream out = new BufferedOutputStream(
                new FileOutputStream(targetFile));
            try {
                int b;
                while ((b = in.read()) != -1)
                    out.write(b);
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }
    ...
}
```

The `saveAction()` method is a JSF action that calls `saveUploadedFile()` and returns the `OK` outcome if no exception is thrown:

```
import javax.faces.application.FacesMessage;
import javax.servlet.ServletException;
import java.io.File;
import java.io.IOException;

public class UploadBean {
    public static final String OK_OUTCOME = "OK";
    public static final String FILE_COMP_ID = "file";
    public static final String CANNOT_OVERRIDE_ERROR
        = "CANNOT_OVERRIDE";
    public static final String IO_EXCEPTION_ERROR
        = "IO_EXCEPTION";
    ...
    public String saveAction() {
        boolean ok = false;
        try {
            File dir = getDataDir();
            if (!dir.exists())
                dir.mkdirs();
            File file = new File(dir, uploadedFile.getFilename());
            if (file.exists() && !override)
                error(FILE_COMP_ID, CANNOT_OVERRIDE_ERROR,
                    FacesMessage.SEVERITY_WARN);
            saveUploadedFile(file);
            fileName = file.getName();
            fileSize = file.length();
            contentType = uploadedFile.getContentType();
            ok = true;
        } catch (IOException iox) {
            try {
                error(FILE_COMP_ID, IO_EXCEPTION_ERROR,
                    FacesMessage.SEVERITY_ERROR);
            } catch (ServletException sx) {
                getServletContext().log(sx.getMessage(), iox);
            }

        } catch (ServletException sx) {
            getServletContext().log(sx.getMessage());
        } finally {
            uploadedFile.dispose();
            uploadedFile = null;
        }
        return ok ? OK_OUTCOME : null;
    }

}
```

Using ADF Faces in JSF Pages

The example Web application contains two JSF pages (UploadForm.jsp and UploadResult.jsp) that are presented in this section.

Building the file-uploading form. The UploadForm.jsp page declares the used JSF and ADF tag libraries, wraps all UI tags within <f:view> as with any JSF page, loads the resource bundle with <f:loadBundle>, and generates the HTML header and the <body> tag with the <afh:html>, <afh:head>, and <afh:body> tags that ADF Faces provides.

The HTML <form> tag is produced by <af:form>, whose `usesUpload` attribute is true, indicating that the encoding type must be `multipart/form-data`. All UI components are placed within <h:panelGrid> that creates an HTML table. The <h:outputText> tag is used to output the page's title, and <af:objectLegend> adds a message that tells the user that an asterisk (*) marks the required fields. The <af:inputFile>, <af:selectBooleanCheckbox>, and <h:commandButton> tags define the main UI components of the page: a file-input element, a checkbox, and a submit button:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="afh" uri="http://xmlns.oracle.com/adf/faces/EA10/html" %>
<%@ taglib prefix="af" uri="http://xmlns.oracle.com/adf/faces/EA10" %>

<f:view>
    <f:loadBundle var="resources" basename="adfupload.Resources"/>
    <afh:html>
        <afh:head title="#{resources.title}"/>
        <afh:body>
            <af:form usesUpload="true">
                <h:panelGrid columns="1" border="0" cellspacing="5">
                    <h:outputText value="#{resources.title}"/>
                    <af:objectLegend name="required"/>
                    <af:inputFile id="file" required="true"
                        value="#{uploadBean.uploadedFile}"
                        label="#{resources.file}"/>
                    <af:selectBooleanCheckbox id="override"
                        value="#{uploadBean.override}"
                        label="#{resources.override}"/>
                    <h:commandButton id="upload"
                        action="#{uploadBean.saveAction}"
                        value="#{resources.upload}"/>
                </h:panelGrid>
            </af:form>
        </afh:body>
    </afh:html>
</f:view>
```

Figure 1 contains the HTML form produced by UploadForm.jsp:

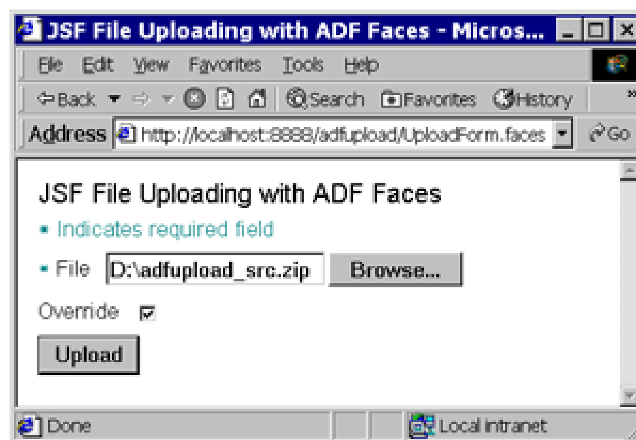


Figure 1: Showing the information about the uploaded file

The UploadResult.jsp page shows the name of the uploaded file, its size, and the content type. This page also provides a link that allows the user to go back to the file-uploading form:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="afh" uri="http://xmlns.oracle.com/adf/faces/EA10/html" %>
<%@ taglib prefix="af" uri="http://xmlns.oracle.com/adf/faces/EA10" %>

<f:view>
    <f:loadBundle var="resources" basename="adfupload.Resources"/>
    <afh:html>
        <afh:head title="#{resources.title}"/>
        <afh:body>
            <h:panelGrid columns="1" border="0" cellspacing="5">
                <h:outputText value="#{resources.title}"/>
                <af:panelLabelAndMessage valign="bottom"
                        label="#{resources.fileName}">
                    <h:outputText styleClass="OraFieldText"
                        value="#{uploadBean.fileName}"/>
                </af:panelLabelAndMessage>
                <af:panelLabelAndMessage valign="bottom"
                        label="#{resources.fileSize}">
                    <h:outputText styleClass="OraFieldText"
                        value="#{uploadBean.fileSize}"/>
                </af:panelLabelAndMessage>
                <af:panelLabelAndMessage valign="bottom"
                        label="#{resources.contentType}">
                    <h:outputText styleClass="OraFieldText"
                        value="#{uploadBean.contentType}"/>
                </af:panelLabelAndMessage>
                <h:outputLink value="UploadForm.faces">
                    <h:outputText value="#{resources.backLink}"/>
                </h:outputLink>
            </h:panelGrid>
        </afh:body>
    </afh:html>
</f:view>
```

The result you see in the Web browser when you upload a file is shown in Figure 2:

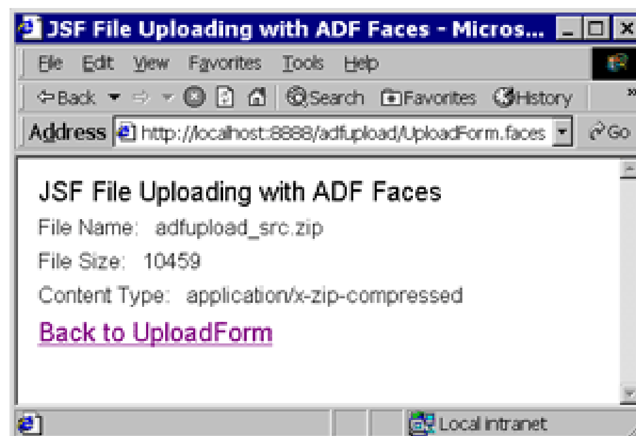


Figure 2: Testing the example application

The source-code archive of this article contains a directory called adfupload_src that has two subdirectories, java and web. The former contains the JavaBean class and the resources bundle, while the latter contains the example Web application without the JAR files of JSF and ADF. The readme.txt file describes how to obtain and where to copy the required JARs.

Conclusion

This article briefly discussed how we decide on which framework to use and also discussed the support for file uploading provided by ADF Faces. Besides this feature, ADF Faces offers many other UI components, including tables and trees, which allow you to focus on your main task when building complex Web UIs. ADF Faces takes care of things such as multipart/form-data parsing, consistent look-and-feel and UI state management, so that you can build Web pages from UI components instead of dealing with HTML tags, JavaScript code, and CSS. And if the tag libraries don't offer everything you need, you can use the APIs provided by JSF and ADF Faces to build custom components that are reusable across applications.

Next Steps

Framework next steps

1) Download the source code of this article's example Web application

2) Get Early Access Release of Oracle ADF Faces Components: a rich set of JSF components.

3) Learn How To Use ADF Faces With JDeveloper 10*g*

Additional Information on Frameworks:

Making Faces
JavaServer Faces (JSF) provides a powerful new framework for creating Web GUIs.

From ADF UIX to JSF
Oracle ADF Faces Components brings a library of reuse to JavaServer Faces.

List of ADF Faces Components

Read the Oracle ADF Faces FAQ

Oracle JDeveloper and JSF

Roadmap for the ADF UIX technology and JavaServer Faces

More on JavaServer Faces Technology

Oracle ADF Development Guidelines

Andrei Cioroianu (devtools@devsphere.com) is the founder of Devsphere (www.devsphere.com): a provider of Java frameworks, XML consulting, and Web development services. Cioroianu has written many Java articles published by Oracle Technology Network, ONJava (www.onjava.com), JavaWorld (www.javaworld.com), and *Java Developer's Journal*. He also coauthored the books *Java XML Programmer's Reference* and *Professional Java XML* (both from Wrox Press).

Copyright © 2005, Oracle. All Rights Reserved.