TIBCO ActiveMatrix® BPM WebApp Component Development

Software Release 3.0
May 2014

Two-Second Advantage®

Important Information

Contents

TIBCO Documentation and Support Services

All TIBCO documentation is available in the TIBCO Documentation Library, which can be found here:

https://docs.tibco.com

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support as follows:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to:

http://www.tibcommunity.com

Overview

WebApp components can be created by bringing in an existing Web application in the TIBCO ActiveMatrix platform.

Introduction

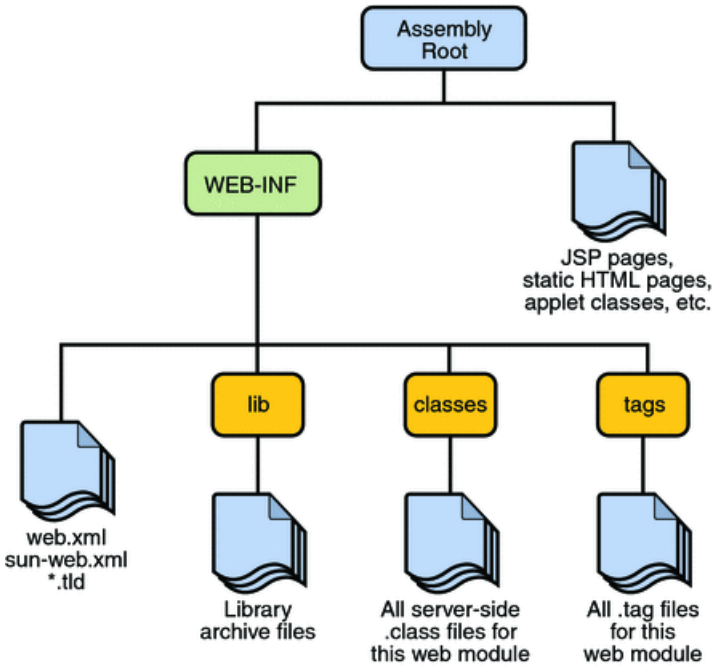
A Web application is a group of HTML pages, JSP pages, servlets, resources and source file, which can be managed as a single unit.

Web applications can be simple (consisting of only static Web pages) or they can be advanced and include JavaServer Pages (JSP) files and Java servlets. During development, these resources, along with an XML deployment descriptor (and other Web resources), are contained within a Web project.

When you are ready to publish the Web application to the Web, you deploy the Web project to the server as a Web archive (WAR) file. The end user can then view the Web application as a website from a URL.

In TIBCO ActiveMatrix Service Grid, all the resources are archived in the distributed application archive (DAA), which then internally deploys the required WAR file.

The structure of a standard web module is shown in the following diagram.



The WebApp component integrates Java EE web applications into TIBCO ActiveMatrix Service Grid and TIBCO ActiveMatrix BPM platform. The integration conforms to the SCA Java EE Integration Specification (https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sca-j).

Approaches

You can create a WebApp component either top-down, bottom-up, by bringing an existing Web application into the TIBCO ActiveMatrix platform.

Top-down approach:

- You can configure the component reference implementation using a WTP (Web Tools Platform) project created in SOA Development Studio (SDS) during Generate Servlet Implementation.
- **Plugin project:** use this option to create an OSGi-enabled web application.

Bottom-up approach: You bring an already existing Web application into the TIBCO ActiveMatrix platform. The existing Web application can be a WAR (Web ARchive) file, WTP project, or an OSGi-fied WebApp. If you start with an existing:

- **WAR file**: you cannot add Properties or References on a component. The DAA has the WAR file bundled in it.
- **WTP project:** you can add Servlets, References, and properties if required. The WTP project is exported into a WAR file and bundled inside the DAA.

Web Application Components

WebApp components can be created, configured, and updated. You can configure a WebApp component's custom feature or external custom feature. You can create a ZeroConfiguration DAA if needed.

Creating a WebApp Component

To create a WebApp component you add an empty WebApp component or use an existing implementation.

Creating an SOA Project

A new SOA project can be created from an existing WAR file or WTP file.

Procedure

1. Select **File > New > TIBCO SOA Resources**.
2. Click **TIBCO SOA Project** and click **Next**.
3. In the **Project Name** field, type a name for the project and click **Next** twice.
4. In the **Project Types** list, choose one of the following ways to create the project:
   a) **SOA Project From Implementation**
      You can create a Web application project from an existing WAR file or WTP file.
   b) **Empty SOA Project**
   c) **Basic SOA Project**
5. Click **Finish**.

Adding an Empty WebApp Component

There are several ways to add an empty WebApp component.
To add an empty WebApp component, do one of the following:

- Right-click the composite canvas and select **Add > WebApp**, or
- Click the canvas and click the **WebApp** icon in the pop-up toolbar, or
- Click the **WebApp** icon in the **Palette** and click the canvas.

Starting With an Existing Implementation

You can create a SOA project from an existing implementation either using a bottom-up approach or a top-down approach.

- **Bottom-up approach:**
  - Method 1: Drag and drop the existing WAR file from the Project Explorer to the composite canvas.
  - Method 2: In the **Properties** tab, select **Implementation** > **Basic**, and specify a WTP project or WAR file. WTP project as the default option.
- **Top-down approach**:
  1. Drag and drop the **WebApp** component on the composite canvas.
  2. Add a reference on the **WebApp** component and attach WSDL port type to the reference.
  3. In the Properties tab, select **WTP Project** or **Plugin Project**.
  4. Select **Generate Servlet Implementation**. If **WTP Project** was selected, a WTP project is generated. If **Plugin Project** was selected, a Plugin project is generated.

OSGi-enabled WebApp Component

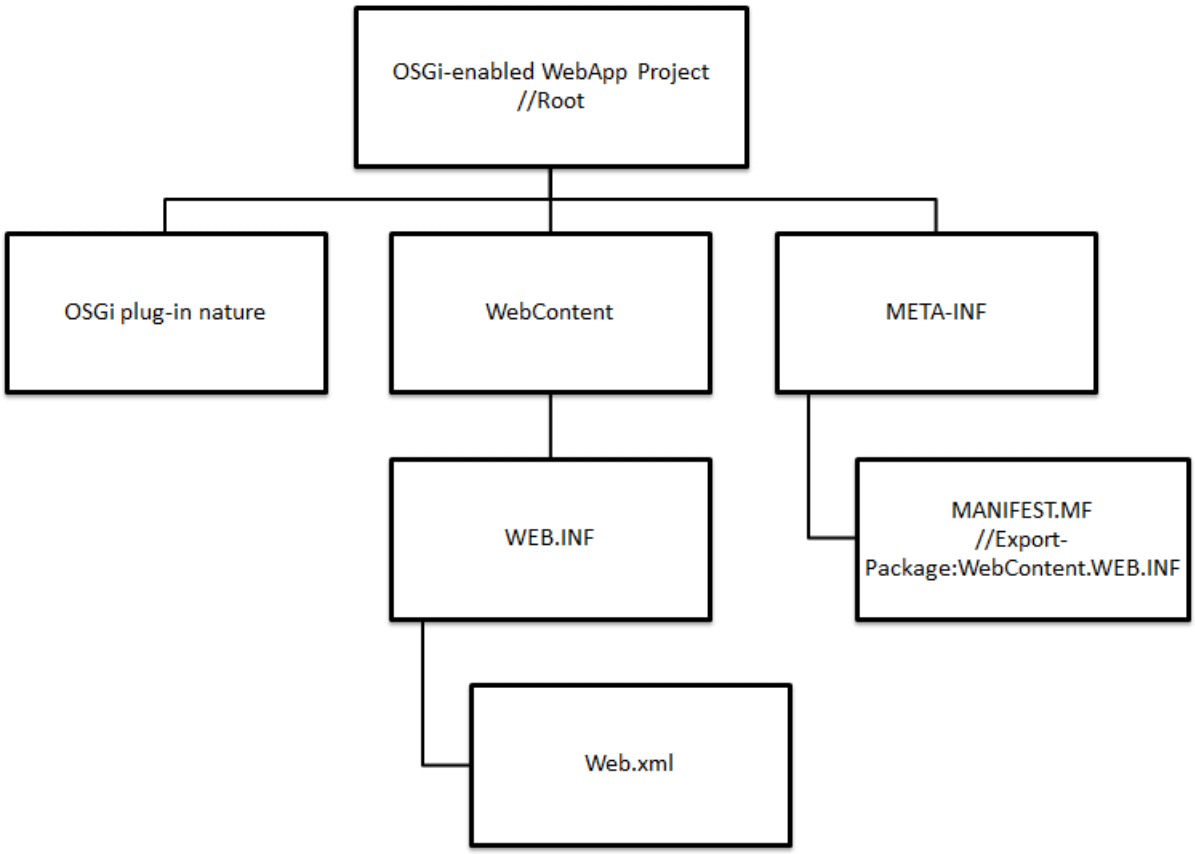
You can create an OSGi-enabled web application by selecting the implementation type as a **Plugin Project**. When the web application is OSGi-enabled, a `web.xml` file is packaged in the OSGi plug-in. The advantages of creating an OSGi-enabled WebApp component are:

- Loads all the resources from the Implementation Bundle (OSGi-enabled WebApp) using the Implementation Bundle Class Loader instead of the Component Bundle Class Loader.
- Removes the overhead of exporting all the resources from the Implementation Bundle (which require user inference and is error prone).
- Behaves as a self-justifying bundle where all the dependencies can be specified in it's own `MANIFEST.MF` file instead of specifying them in the component's `.requirement` file.

Structure of Bundle

The WebApp component is composed as a set of OSGi bundles. OSGi bundles are JAR files that typically contain Java class files of the service interfaces, their implementation, and some meta information in a `META-INF/manifest.mf` file. Services are Java interfaces. After the bundle is registered as a service with the OSGi framework, other bundles can use the "published" service. You can add Servlets, References, and properties if required. The Servlets and all static content (HTML, JPEG files, and so on) reside in Java plug-ins.

The typical directory structure is shown below:



- Static resources such as HTML files or image files can directly be placed in the `WebContent` directory or in a sub-directory.
- All dependencies of the Implementation Bundle needs to be specified in it's own `MANIFEST.MF` file instead of a `.requirement` file.
- The WebApp component has a **require-bundle** dependency to it's implementation.

Web.xml File

The `web.xml` file is generated using a **Generate Servlet Implementation** wizard. This wizard generates a default `web.xml`. Using any other existing `web.xml` is not allowed. The default values in `web.xml` are inline with the **Generate WTP Wizard** provided by Eclipse. The default `web.xml` is of version 2.4. Selecting different versions of `web.xml` while generating the implementation is not supported. If required, you need to manually change the version of `web.xml`.

Creating an OSGi-enabled WebApp Component

Procedure

1. Create an empty SOA project as described in Creating an SOA Project on page 8.
2. Create an empty WebApp component as described in Adding an Empty WebApp Component on page 8.
3. In the **Properties** tab, select **Plugin Project**.

4. Select the WebApp component created, right-click, and select **Generate Servlet Implementation**. After the implementation is generated successfully, the:
   - **Plugin Project** and **Deployment Descriptor (web.xml) File** fields are populated.
   - Implementation bundle includes the `web.xml` file (see the Project Explorer view).
   - `MANIFEST.MF` file is created (see the Project Explorer view).

Configuring a Web Application Component

You can configure a WebApp component's custom feature or external custom feature. You can also use the ZeroConfiguration WAR file.

Configuring a WebApp Components Custom Feature

One option to configure a WebApp component is to use the custom feature.

When you generate a Web application whose implementation type is **Plugin Project**, a custom feature is automatically created and configured. A custom feature is not automatically created if you choose the WTP and WAR file options. For more information on custom features, see *TIBCO ActiveMatrix Java Component Development.*

If you manually configure the component's implementation, you must manually create and configure the custom feature by selecting **File > New > Other > Custom Feature**. If the component implementation uses a library, add the custom feature containing the library in the **Properties** view.

Procedure

1. In the **Properties** view, click the component.
2. Click the **Implementation** tab.
3. Click the  button to the right of the Features tables.
   The **Select a Feature** dialog displays.
4. In the **Select an item to open** field, type a partial feature name.
   The feature that matches the name displays in the **Matching items** list.
5. Click a feature and click **OK**.
   The feature is added to the **Features** list.

Configuring a WebApp Components External Custom Feature

One option to configure a WebApp component is to use the external custom feature.

If your WebApp component implementation references a plug-in containing a shared library, you must add the custom feature that contains the plug-in to the WebApp component's configuration. For more information on custom features, see *TIBCO ActiveMatrix Java Component Development.*

Procedure

1. Click the component.
2. In the **Properties** view, click the **Implementation** tab.
3. Click the  button to the right of the **Features** table.
   The **Select a Feature** dialog displays.
4. Click **OK**.
   The feature is added to the component's **Features** list.

WebApp Component Reference

Field	Description
WTP Project	Start with the Eclipse WTP project containing the WebApp component implementation.
WAR	Start with the WAR file containing the WebApp component implementation.
Plugin Project	Creates a plugin project implementation.

Properties

The Context Root and Connector Name properties must always be set.

Field	Description
contextRoot	The context root of a web application determines which URLs are to be delegated to your web application. If your application's context root is myapp, any request for `/myapp` or `/myapp/*` are handled. For example, `http://localhost:8080/myapp/index.html`. **NOTE:** • A WebApp component must have a unique context root. • The contextRoot property must have only one element.
defaultConnector	This property defines the name for an HTTP Inbound connector. For a Web application, a browser is only way of communication and as browser uses HTTP to communicate with any Web application. In TIBCO ActiveMatrix, you need to configure this HttpInbound Resource template in Administrator before deploying a Web application.

Compute Feature Dependencies

Field	Description
Compute Feature Dependencies	Indicate whether to compute the features on which the component bundle depends. When unchecked, the Feature Dependencies table displays. Default: • New projects - selected. • Legacy projects - cleared.
Preview	Displays a dialog containing a list of features on which the component bundle depends.

Features Dependencies

Column	Description
Feature ID	ID of the feature.
Version Range	Range of feature versions.

By default, the table lists the details of the automatically-generated feature containing the component implementation bundle.

Plugin Project

Field	Description
Plugin Project	Selected plugin project implementation.
Deployment Descriptor (`web.xml`) File	Location of the `web.xml` file.
Thread Context Class Loader Type	Configures the Thread Context Class Loader property: • component - The class loader of the component bundle. The class loader has visibility to the component bundle class path space, Import-Package, and Require-Bundle entries from the component. • bundle - The class loader of the implementation bundle. The class loader has visibility to the bundle class path space and the Class-Space because of entries in the `MANIFEST.MF` file. • none - A null thread context class loader. Default: component

Package the Implementation Bundle With the Application

Field	Description
Package the implementation bundle with the application	Indicate whether to compute the component bundle dependencies. When a component is deployed on a node, ActiveMatrix generates a component bundle. When selected, the component implementation bundles required by the component bundle are computed and identified when you package the composite. When cleared, the Implementation Dependency and Compute Feature Dependencies fields display and you can manually specify the dependencies. Default: • New projects - selected. • Legacy projects - cleared.

Field	Description
Implementation Dependency	Type of the dependency of the component bundle on the component implementation. • Require Bundle - The bundle containing the component implementation is declared as a required bundle. When selected, the Bundle Name field displays. Default: Require Bundle
Bundle Name	Symbolic name of the bundle containing the component implementation. Default: The bundle in which the component implementation class is present.
Package Name	Name of the package containing the component implementation. Default: The package in which the component implementation class is present.
Version Range	Versions of the bundle or package that satisfy the component bundle's dependency. When specifying a range for a bundle, you may require an exact match to a version that includes a build qualifier. In contrast, the range for a package is inexact. Default: • Bundle - [1.0.0.qualifier,1.0.0.qualifier]. • Package - [1.0.0, 2.0.0).

Adding Configuring a WebApp Components Security

Resources of a Web application are secured using security policies that provide authentication, access control for resources, and confidentiality or data privacy.

Authentication: The means by which communicating entities prove to one another that they are acting on behalf of specific identities authorized for access.

Access control for resources: The means by which interactions with resources are limited to collections of users or programs in order to enforce integrity, confidentiality, or availability constraints.

Confidentiality or data privacy: The means used to ensure that information is made available only to users who are authorized to access it.

The WebApp component provides the Form-based Authentication and the Security Constraint policies to implement security policies for authentication and authorization of resources.

If a WebApp component is created from a WAR file or WTP project, which already contains the security configuration in `web.xml`, the security configuration from `web.xml` will be mapped to the WebApp's policy configuration.

Do not add or modify the form-based authentication data directly in `web.xml`. You must use the provided interface (Implementation > Security tab or Policies tab) to do this.

Using Form-based Authentication Policy

The authentication mechanism provides the means for verifying user access to the website's protected area, based on user name and password. The form-based authentication mechanism lets you set up the look and feel of login as well as error screens.

Login screens present a form to enter username and password while accessing a protected resource. The login module checks user authority to access the resource. If the user is not authenticated, the error page is returned.

Form-based login uses sessions for login. The system automatically logs out a user from the application if the session is invalidated.

Adding Configuring Form-based Authentication Policy

WebApp components can be configured for form-based authentication.

Procedure

1. Select the WebApp component you need to configure in the editor.
2. Select the **Properties** view and use either of the following approaches to open the Form-based Authentication Configuration wizard window:
   - Approach 1: In the **Properties** view, click the **Implementation** vertical tab and select the **Security** tab. Under the **Authentication** section, select **Form** as authentication type from the drop-down.
   - Approach 2: In the **Properties** view, click the **Policies** vertical tab and click the **Add Policy Set** icon. Select **Embedded** as the Policy Set type, and **Form-Based Authentication Policy** under the System Policies list and click **Next**.
3. In the **Form-based Authentication Configuration** wizard window, specify the following parameters:
   a) **Login page**. This page contains fields for entering username and password. Click **Browse** to select the desired login page from the project resource list and click **OK**.
   b) **Error page**. This page displays if authentication fails. Click **Browse** to select the desired error page from the project resource list and click **OK**.
   c) **Login module.**. Resource instance for LDAP configuration.
4. Click **Finish**.

Security Constraint Policy

A security constraint associates authorization and/or user data constraints with HTTP operations on web resources. A Security Constraint policy allows you to set security constraints on one or more web resource collections.

A security constraint, which is represented by `security-constraint` in the deployment descriptor, consists of two main elements:

- **Web resource collection.** The HTTP operations and web resources to which a security constraint applies (i.e., the constrained requests) are identified by one or more web resource collections (`web-resource-collection` in the deployment descriptor). A web resource collection consists of URL patterns (`url-pattern` in deployment descriptor), and HTTP methods (`http-method` in deployment descriptor).
- **Authorization constraint.** An authorization constraint (`auth-constraint` in the deployment descriptor) establishes a requirement for authentication, and names the authorization roles permitted to perform the constrained requests. A user must be a member of at least one of the named roles to be permitted to perform the constrained requests. An authorization constraint consists of the role name element (`role-name` in deployment descriptor).

The special role name "*" is a shorthand for all role names defined, while an authorization constraint that names no roles indicates that access to the constrained requests is not permitted under any circumstances.

Security Constraint Definition Example

The following is sample web.xml code to define a security constraint.

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_2_5.xsd"
version="2.5">
<display-name>Test WebApp</display-name>
<security-constraint>
        <web-resource-collection>
            <web-resource-name>Admin Role</web-resource-name>
            <url-pattern>/dump/auth/admin/*</url-pattern>
            <url-pattern>*.htm</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>content-administrator</role-name>
        </auth-constraint>
</security-constraint>
<security-constraint>
        <web-resource-collection>
            <web-resource-name>Super User Role</web-resource-name>
            <url-pattern>/dump/auth/admin/*</url-pattern>
            <url-pattern>/dump/auth/display/*</url-pattern>
            <http-method>HEAD</http-method>
        </web-resource-collection>
        <web-resource-collection>
                    <web-resource-name>Super User Role</web-resource-name>
                    <url-pattern>/dump/auth/system/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                    <role-name>Super-User</role-name>
        </auth-constraint>
    </security-constraint>
</web-app>
```

Adding Configuring A Security Constraint Policy

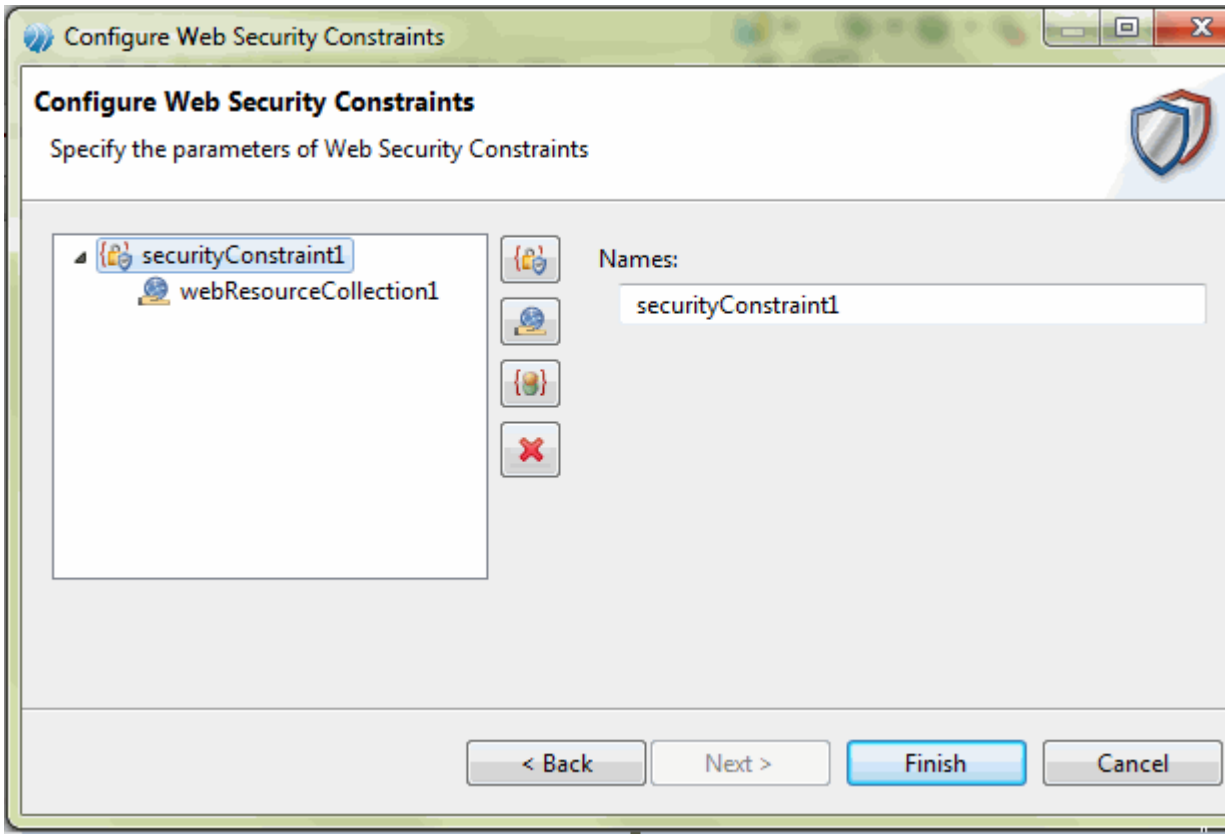
In TIBCO Business Studio you can add/configure a security constraint policy using a wizard.

Procedure

1. Open the **Configure Web Security Constraint** wizard using either of the following approaches:
   - **Approach 1**: In the **Properties** view, click the **Implementation** vertical tab and select the **Security** tab.

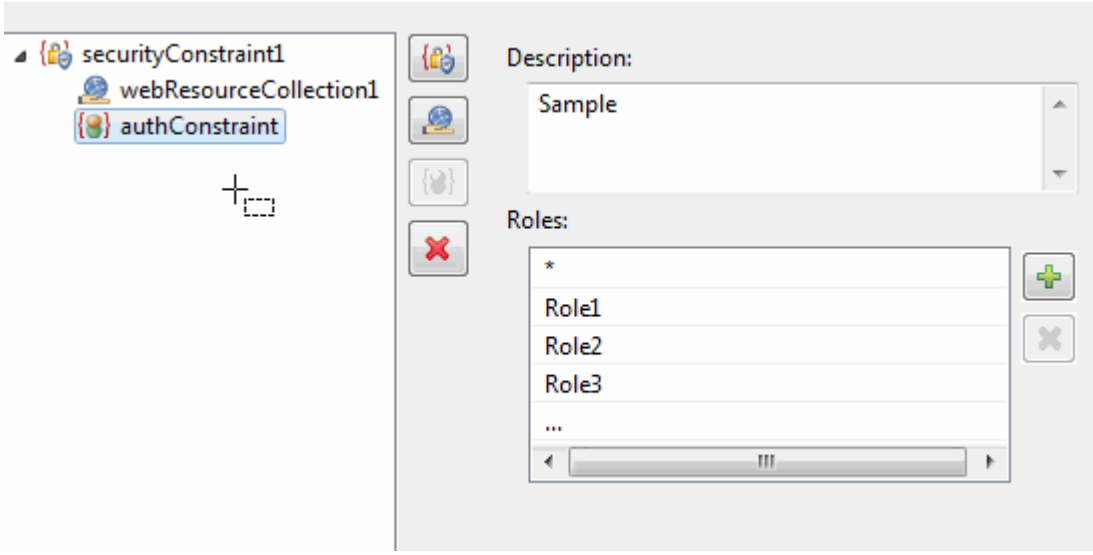
     Under the **Web Security Constraint** section, either click the policy set to configure it, or click the -not set- hyperlink to add a new Web Security Constraint policy set.
   - **Approach 2**: In the Properties view, click the **Policies** vertical tab, and click the **Add Policy Set** icon.

     Select **Embedded** as the Policy Set type, **Web Security Constraint Policy** under the System Policies list, and click Next.



2. Select the security constraint and rename the security constraint if required.
3. Click on the add web resource collection icon  to add web resources under the security constraint or select existing web resource collection to update.
4. Type the **Name** and **Description** of the web resource collection.
5. In the URL pattern field, click the plus icon to add a URL pattern. Double-click on a listed URL pattern to modify it.
6. Check the HTTP methods to be allowed for the web resource collection. Default is "all" HTTP methods. If no methods are selected then "all" HTTP methods option is checked.
7. Click the add auth-constraint icon  to add a authorization constraint.

8. Type the **Description**.
9. Click on plus icon to add a role name to the permissible roles list for the security constraint. Click on a listed role name to modify the role name.



10. Click the add security constraint icon  to add another security constraint to the policy and repeat steps 2 through step 9.
11. Click **Finish** when you are done updating the security constraint policy.

Updating a WebApp Component

You can update a component after you have configured its implementation.

Follow the steps in the table below to update a component.

Control	Procedure
Canvas	Right-click the component and select **Refresh from Implementation**.
Canvas	Right-click the component and select **Quick Fixes > Update Component from Implementation**. (The "**Quick Fixes**" option is available when you add, delete or update a service, reference and property from the WebApp component.)
Properties View	1. Select **Properties > General > Validation Report** and click the **fix**... link. 2. Select **Generate Servlet Implementation**.
Problems View	1. In the **Problems View**, right-click an error of the form "`The component <ComponentName> is out of sync with its implementation`" and select **Quick Fix**. (The "**Quick Fix**" option is available when you add, delete or update a service, reference and property from the WebApp component.) 2. In the **Quick Fix** dialog select **Update Component from Implementation**. 3. Click **Finish**.

ZeroConfiguration DAA Creation Using WAR

ZeroConfiguration DAA creation supports existing WebApps that do not invoke an SCA reference. ZeroConfiguration DAA creation is based on SDS command-line support.

You can create a DAA using existing WebApp components (WAR files) on the ActiveMatrix platform without using the composite editor. For details, refer to *SDS Commandline help*.

You must not bundle any of the following files inside the WAR file: `WEB-INF/lib`: `j2ee.jar, jasper-*.jar, jsp-api.jar, rt.jar, tools.jar, servlet.jar, servlet-api.jar, xerce.jar, xerces.jar,` or `xercesImpl.jar.`

Limitations on WAR Files

If the Web application code in the WAR file uses APIs from the following packages, perform the steps listed in this section.

- `javax.xml.*`
- `org.xml.*`
- `org.w3c.*`
- `org.apache.commons.logging.*`
- `org.apache.log4j.*`

Imports in .requirements File for Necessary APIs

Make sure that all the packages and sub-packages from the above list are declared as imports in the `.requirements` file of the WebApp IT component.

For example, if the web application uses the `javax.xml.transform` package, add it in the `.requirements` file as follows:

1. Search for the required package using the Plugin Registry View in TIBCO Business Studio.
2. Override the `.requirements` file of the WebApp IT component and add the necessary import package entry.

Remove API implementation from WAR

When any of the above mentioned APIs are being imported, remove the conflicting implementation JARs from the WAR (using a tool such as 7-zip).

For example, let us say, your application uses the JAXP APIs. You have added the necessary import packages in the `.requirements` file for the API packages. The WAR file bundles the Apache Xalan JAR file that provides the implementation of these APIs. In this case, you need to remove the Xalan JAR from the WAR's `lib` folder.

Declaring Dependencies on org.ietf.jgss Packages

Normally, if you import packages and do not add them to the manifest, TIBCO Business Studio displays an error. However, if you import any of the `org.ietf.jgss` packages and do not declare the import in the manifest, TIBCO Business Studio does not display an error because TIBCO Business Studio resolves those packages from the configured JRE. If you then deploy the application without the declaration in the manifest, the application will not run. Hence, you must ensure that you import the `org.ietf.jgss` package in the manifest file.

Web Application Component Implementations

WebApp component implementations can be generated, regenerated, and refreshed. You can access properties and invoke reference options, as well as test your components.

Opening an Implementation

For WTP and Plugin Project options, the `web.xml` file is opened. For a WAR file, the WAR file is just highlighted in the Project Explorer.

The following table explains how to open an implementation.

Control	Procedure
Canvas	Double-click the component.
Project Explorer	Select the WTP project and open the implementation.
Canvas	Right-click the component and select **Open Implementation**.

Generating an Implementation

You generate an implementation by generating the servlet and then using the **Code Generation Details** dialog.

Procedure

1. Select **Properties General > Validation Report**, and click the **fix...** link.
2. Click **Generate Servlet Implementation**.
3. Using the **Canvas** control, right-click the component and select **Quick Fixes > Generate Servlet Implementation.**
4. Right-click the component and select **Generate Servlet Implementation**.
5. In the **Problems** view, right-click an error of the form "`Component <ComponentName> is not configured`" and select **Quick Fix**.
6. In the **Quick Fix** dialog, click **Generate Servlet Implementation**.
7. Click **Finish**.
   The Code generation details dialog displays.
8. Complete the process described in Generate WebApp Component Implementation.
9. Click **Finish**.
   A WTP implementation is generated.

   The `WebContent` folder contains items to be published to the server. By default, this folder will be named `WebContent` for newly created static and dynamic Web projects.

   - `META-INF` — This directory contains the `MANIFEST.MF` file, which is used to map class paths for dependent JAR files that exist in other projects in the same Enterprise Application project. An entry in this file will update the run-time project class path and Java build settings to include the referenced JAR files.
   - `WEB-INF` — The directory where supporting Web resources for a Web application are kept (for example: .xmi files, `.xml` files, and `web.xml`.)

Generate WebApp Component Implementation

To generate a WebApp component implementation refer to the following tables describing the **Code Generation Details** dialog, the **XML Data Binding Classes** dialog, and the **Create Servlet** dialog.

Code Generation Details Dialog

Refer to this table when generating a WebApp component implementation.

Field	Description
Project	The name of the web application project to contain the implementation. Default: • For WTP: "`WebApp`" + <*name of composite*> • For Plugin Project: "`com.webapp`" + <*name of composite in lower case*>
Source Location	The name of the source folder in the plug-in project. Default: `src`
Package	The name of the package of the implementation.
Class	The name of the class of the implementation. Default: The name of component is the default class name.
Use default location for generated superclass	Default: checked.
Superclass package	The name of the package of the abstract superclass of the implementation class.
Superclass class	The name of the abstract superclass of the implementation class. Default: `Abstract`<*WebappComponentName*>

Normally, if you import packages and do not add them to the manifest, TIBCO Business Studio displays an error. However, If you import any of the `javax.xml.*` or `org.ietf.jgss` packages and do not declare the import in the manifest, TIBCO Business Studio does not display an error because TIBCO Business Studio resolves those packages from the configured JRE. If you then deploy the application without the declaration in the manifest, the application will not run. Hence, you must ensure that you import `javax.xml` or `org.ietf.jgss` packages in the manifest file.

XML Data Binding Classes Dialog

Refer to this table when generating a WebApp component implementation.

The **XML Data Binding Classes** dialog appears if the WebApp component is wired to any reference.

WebApp component supports JAXB based code generation. For details, refer to *TIBCO ActiveMatrix Java Component Development.*

Field	Description
Type	The type of the data binding being generated: XMLBeans or JAXB. If a JAR file already exists for the contract selected in the **Contracts** list, and you choose a binding type different than the one that exists in the JAR file, or the contract has changed since the JAR file was generated, the **Overwrite Existing JAR** checkbox will be checked. Default: XMLBeans.
Contracts Details	
Contracts	A list of WSDL and schema files for which XML data binding classes will be generated.
JAR Type	The type of JAR file being generated: Beans or Interface. (read only)
Source File	The path to the source file containing the selected contract. (read only)
JAR File	The path to the JAR file. Default: When generating a component implementation: • Beans – For a Plugin Project: *projectName*/libs/contractFileName.wsdl.jar – For a WTP Project: /projectName/WebContent/WEB-INF/lib/contractFileName.wsdl.jar • Interface – For a Plugin Project: projectName/libs/*contractFileName*.wsdl_interface.jar – For a WTP Project: /projectName/WebContent/WEB-INF/lib/*contractFileName*.wsdl_interface.jar Where *contractFileName* is the name of the file containing the contract selected in the **Contracts** list and *projectName* is the name of the project containing the component implementation.
Set JAR Destination Folder	Invokes a dialog where you can set the folder to contain generated JAR files: • All Generated JARs - All JAR files will be generated in the same folder as the destination of the currently selected JAR. • New Generated JARs - Only newly generated JAR files will be generated in the same folder as the destination of the currently selected JAR file. Setting the JAR folder affects only the JAR files generated by the wizard. It has no effect outside the wizard nor on subsequent wizard runs. Default: All Generated JARs.

Field	Description
JAR Status	The status of the JAR file containing the classes generated for the selected contract: • JAR is non-existent and will be generated. - The JAR file does not exist. • Different binding type. JAR must be overwritten. - The value of the **Type** field is different than the type of the data binding classes in the JAR file. • JAR exists and will be overwritten. - The JAR file exists and the **Overwrite Existing JAR** checkbox is checked. • JAR exists and will be preserved. - The JAR file exists and the Overwrite Existing JAR checkbox is unchecked. • JAR is outdated and will be overwritten. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is checked, so the JAR file will be generated. • JAR is outdated and will be preserved. - The selected contract has changed since the JAR file was generated and the Overwrite Existing JAR checkbox is unchecked, so the JAR file will not be generated.
Overwrite Existing JAR	Enabled only when the JAR file exists. When checked, the JAR file will be regenerated. When unchecked, the existing file will be reused and will not be modified.
Advanced	
Use Configuration File	Indicate that the specified data binding configuration file should be used when generating JAR files. When you check the checkbox, the text field is enabled. Default: Unchecked.

Create Servlet Dialog

Refer to this table when generating a WebApp component implementation.

Field	Description
Name	Name of the servlet
Description	Description of the servlet
Initialization Parameters	Name-value initialization parameters are used to convey setup information. Typical examples are a Webmaster's e-mail address, or the name of a system that holds critical data.
URL Mappings	Upon receipt of a client request, the URL mappings determine the Web application to which to forward it. For more details, see URL Mappings.

Regenerating an Implementation

You can regenerate an implementation without recreating everything. For example, if you have a Web application with a Java SOA project, each can have their respective implementations.

After developing the component, if you need to make a change to the WSDL (for example, a change to the datatype), you can use this option to change the existing implementation to apply the updated WSDL instead of creating everything from scratch again.

The implementation must have been originally generated before you can regenerate.

You should regenerate the component implementation after you add (or delete) a service, reference, or property to the component.

Control	Procedure
Canvas	Right-click the component and select **Regenerate Servlet Implementation**.
Problems View	1. In the **Problems** view, right-click an error of the form "`The component <ComponentName> is out of sync with its implementation`" and select **Quick Fix**. 2. In the **Quick Fix** dialog select **Update Component from Implementation** or **Update/Create Servlet**. 3. Click **Finish**. The implementation is updated to match the component.

Refreshing an Implementation

This option updates the SDS WebApp component based on an underlying implementation. For example, a WebApp component is configured with two properties and an implementation is generated. If one of the properties is accidentally deleted, you can use the Refresh option. The SDS component reads the underlying implementation and refreshes the UI with the two properties.
The following table explains how to refresh an implementation.

Control	Procedure
Canvas	Right-click the component and select **Refresh from Implementation.**
Problems View	1. In the **Problems** view, right-click an error of the form "`The component <ComponentName> is out of sync with its implementation`" and select **Quick Fix**. 2. In the **Quick Fix** dialog select **Update Component from Implementation** or **Update/Create Servlet**. 3. Click **Finish**.

Accessing a Property

When you generate a WebApp component implementation for a component with a property, TIBCO Business Studio adds a field that represents the property and accessor methods to the WebApp component's abstract implementation servlet.

See *TIBCO ActiveMatrix Java Component Development* for details.

Invoking a Reference Operation

When you add an In-* reference to a WebApp component, a field and accessor methods are added to the component.

See *TIBCO ActiveMatrix Java Component Development* for details.

WebApp Component Testing

When AMX composite applications run in RAD, or a remote admin is connected through TIBCO Business Studio, you can view information about the WebApp components using the internal WebApp component testing servlet.

The WebApp component testing servlet launches the OSGi-based Jetty server that hosts the WebApp component to be tested. It launches the WebApp component in the Eclipse internal browser.

To test the WebApp component, right click the WebApp component running in the Administrator Explorer view, and select **Invoke WebApp with Web Explorer**.

The application detects if the WebApp component is running in RAD environment or at remote machine (JAD environment). Based on the communication environment, the WebApp component is processed.

RAD Communication

If the WebApp component is running in RAD, the launch configuration is resolved as follows:

Procedure

1. The launch configuration for the web application DAA or composite file is resolved.
   - If launched from DAA, the web.xml file is extracted from the composite file.
   - If launched from the composite file, web.xml is resolved from the Eclipse workspace.
2. The `contextRoot` and `defaultConnector` property values are extracted from the composite resource. The HTTP port number associated with the `defaultConnector` property is then extracted from the Debug/RunConfiguration > Advanced > HTTP Connectors section.
3. A test URL using `contextRoot` and HTTP port number (http://localhost:*port number*/*contextRoo*t) is constructed. It opens it in the Eclipse internal browser.
   - If `contextRoot` is mapped to a welcome page, the browser displays the welcome page.
   - If `contextRoot` is *not* mapped to a welcome page, the browser loads the RAD testing page, which displays all the servlets and their mappings from `web.xml`. Click on any servlet link in the RAD testing page to load the associated web page in other frame.

JAD Communication

If the web application runs in a remote machine (JAD environment), administrator web services are invoked using SOAP requests to retrieve the properties for the WebApp component.

Procedure

1. The values of the `contextRoot` and `defaultConnector` properties are extracted from the property map. If the properties are substitutable, the composite properties are resolved from administrator. This is a recursive process until the final value of the substitution is not found.
2. The HTTP host address and HTTP port number are extracted from `defaultConnector`. Then, the following information is retrieved:
   a) Node on which component is running
   b) Resource instance of the `defaultConnector` on the node
   c) Resource template for the resource instance
   d) HTTP port number from the resource template
   e) HTTP host address from the node.
3. A test URL is constructed using `contextRoot`, the HTTP port number, and the HTTP host address (http://*host address:port number*/*contextRoot*). It is opened in the Eclipse internal browser.

   - If `contextRoot` is mapped to a welcome page, the browser displays the welcome page.
   - If `contextRoot` is not mapped to a welcome page, the browser displays the Page Not Found error page.

Logging

TIBCO ActiveMatrix Service Grid supports logging to standard out and using a logging API. For simple demonstration applications, you can log to `SDTOUT`. However, for product applications you should use the logging API.

See *TIBCO ActiveMatrix Java Component Development* for details.

Handling Errors

The WebApp component handles errors in the same way as the Java component. See *TIBCO ActiveMatrix Java Component Development* for details on handling declared and undeclared faults.

URL Mappings

The path used for mapping to a servlet is the request URL from the request object minus the context path and the path parameters. The URL path mapping rules follow a prescribed order.

There are explicit mappings, and in certain cases implicit mappings are allowed.

Use of URL Paths

Upon receipt of a client request, the Web container determines the Web application to which to forward it. The Web application selected must have the longest context path that matches the start of the request URL.

The matched part of the URL is the context path when mapping to servlets. The Web container next must locate the servlet to process the request, using the path mapping procedure described below.

The path used for mapping to a servlet is the request URL from the request object, minus the context path and the path parameters. The URL path mapping rules below are followed in sequence. The first successful match is used with no further matches attempted.

Procedure

1. The container looks for an exact match of the path of the request to the path of the servlet. A successful match selects the servlet.
2. The container recursively attempts to match the longest path-prefix. This is done by stepping down the path tree a directory at a time, using the '/' character as a path separator. The longest match determines the servlet selected.
3. If the last segment in the URL path contains an extension (for example, jsp), the servlet container tries to match a servlet that handles requests for the extension. An extension is defined as the part of the last segment after the last '.' character
4. If neither of the previous three rules result in a servlet match, the container tries to serve content appropriate for the resource requested. If a "default" servlet is defined for the application, it will be used. The container must use case-sensitive string comparisons for matching.

Specification of Mappings

In the Web application deployment descriptor, the following syntax is used to define mappings.

- A string beginning with a '/' character and ending with a '/*' suffix is used for path mapping.
- A string beginning with a '*.' prefix is used as an extension mapping.
- A string containing only the '/' character indicates the "default" servlet of the application. The servlet path is the request URI minus the context path, and the path info is null.
- All other strings are used for exact matches only.

Implicit Mappings

If the container has an internal JSP container, the `*.jsp` extension is mapped to it, allowing JSP pages to be executed on demand. This mapping is termed an implicit mapping. If a `*.jsp` mapping is defined by the Web application, its mapping takes precedence over the implicit mapping.

A servlet container is allowed to make other implicit mappings as long as explicit mappings take precedence. For example, an implicit mapping of `*.shtml` could be mapped to include functionality on the server.

Example Mapping Set

Consider the following set of mappings:

Path Pattern	Servlet
`/foo/bar/*`	servlet1
`/bar/*`	servlet2
`/catalog`	servlet3
`*.bop`	servlet4

The following behavior would result:

Incoming Path	Servlet Handling Request
`/foo/bar/index.html`	servlet1
`/foo/bar/index.bop`	servlet1
`/bar/index.bop`	servlet2
`/catalog`	servlet3
`/catalog/index.html`	“default” servlet
`/catalog/racecar.bop`	servlet4
`/index.bop`	servlet4

In the case of `/catalog/index.html` and `/catalog/racecar.bop`, the servlet mapped to "`/catalog`" is not used because the match is not exact.