IBM ILOG JViews TGO V8.6

Building Web applications

Copyright

Copyright notice

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at *http://www.ibm.com/legal/copytrade.shtml*

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further copyright information see ***<installdir>*** **`/license/notices.txt`**.

Table of contents

Introducing JViews TGO faces components

Core JViews Faces

The IBM® ILOG® JViews TGO JavaServer™ Faces (JSF) solution is a set of faces components that allow you to build JavaServer™ Pages (JSP™) in order to display and interact with network and equipment views in a web application. It is composed of a JSF tag library, the corresponding Java™ classes, and JavaScript™ objects for rendering the business data in the web application.

The IBM® ILOG® JViews TGO Faces extend the JViews Framework Faces, which are themselves based on the core JViews Faces.

The core JViews Faces constitute the lower-level library that depends only on JSF. They define some basic faces components and provide the infrastructure for the other JViews JSF libraries. These components are the following:

- `debugDependencies`
- `messageBox`
- `imageButton`
- `menu`
- `menuItem`
- `menuSeparator`

JViews Framework Faces

The JViews Framework Faces constitute the mid-level library that is based on the core JViews Faces. They add framework capabilities that allow you to implement a view, an overview, interactors, among others. These components are the following:

- `view`
- `overview`
- `zoomTool`
- `panTool`
- `zoomInteractor`
- `panInteractor`
- `mapInteractor`
- `mapRectInteractor`
- `objectSelectInteractor`
- `objectSelectRectInteractor`

- `contextualMenu`

JViews TGO Faces

The JViews TGO Faces constitute the top-level library that extends the JViews Framework Faces. They allow you to customize the view and interactor for JViews TGO and add data source support. These components are the following:

- `networkView`
- `equipmentView`
- `dataSource`
- `selectInteractor`
- `clientSelectInteractor`
- `selectionManager`

The network view faces component

Explains how to build and interact with a network faces component.

In this section

Declaring a network view faces component
Describes how to declare a network view faces component.

Configuring a network view faces component
Explains how to configure the rendering of a network faces component.

Network view component services
Presents the services that are fully compatible.

Declaring a network view faces component

The network view faces component displays the contents of an `IlpNetwork` in a JavaServer™ Page (JSP™) compliant with the JavaServer Faces (JSF) technology. It is implemented by the class `IltFacesNetworkView` and acts as a facade to an `IlpNetwork` component. It provides a convenient API for the most common uses of the network component, such as setting or retrieving the associated data source, accessing the underlying network component, or accessing the network view directly.

JViews TGO faces components are declared in a tag library descriptor (`.tld`) file named `jviews-tgo-faces.tld` that is included in the `jviews-tgo-all.jar`. The JViews TGO faces tag library must be declared in the JSP page before any of its components are used.

How to define the JViews TGO faces tag library and prefix in a JSP page

The declaration is done at the beginning of the JSP file as follows:

```
<%@ taglib uri="http://www.ilog.com/jviews/tlds/jviews-tgo-faces.tld"
prefix="jvtf" %>
```

This statement declares the `jviews-tgo-faces.tld` tag library within a JSP page, and binds all its components to the `jvtf` prefix. Once this is done, you can declare the network view component as follows:

How to declare a network view faces component

```
<jvtf:networkView id="myNetwork"
                    context="#{myContext}"
```

The `networkView` component requires two mandatory tag attributes:

- `id` (component unique identifier): Can be any given string that uniquely identifies this component within a server session.
- `context` (the `IlpContext` to be used): Should be a value binding to an instance of `IlpContext` declared as a managed bean. A default implementation is available for convenience (ilog.tgo.faces.service. `IltFacesDefaultContext`).

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example1.jsp* .

You will find more information about the sample web application in ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring a network view faces component

Explains how to configure the rendering of a network faces component.

In this section

Configuring the client and server side of the networkView component
Describes the tag attributes defined for the networkView component.

Connecting a business data source
Explains the different ways to configure a data source within the network faces component.

Customizing the underlying IlpNetwork component
Describes how to customize the way the underlying IlpNetwork component is created.

Combining faces components
Describes how to connect components from the core JViews Faces and JViews Framework Faces libraries to the network view.

Interacting with the network view component
Describes how to declare predefined interactors and connect them to the networkView component.

Zoom constraints
Describes how to specify zoom levels.

Controlling the displayed area
Describes how to control the area displayed on the client.

Adding pop-up menus
Explains how to define pop-up menus by means of the contextualMenu tag.

Tiling
Describes the tiling support provided by the Network Faces component.

Managing the session expiration
Explains how to manage the user session expiration.

Configuring the client and server side of the networkView component

To display a network view in the client application, you need business data and rendering information that defines how to display these data.

The configuration for the rendering is split into two distinct groups:

- client-side configuration: HTML configuration stored in the DHTML page and the JavaScript™ objects
- server-side configuration: stored in the network faces implementation or in the image servlet

Client-side configuration relates to the behavior and look of the faces component itself. Server-side configuration relates to the network model and the way the representation objects are displayed and laid out.

There are many ways to configure the client and server sides of the `networkView` component. In general, the client-side configuration is passed as tag attributes to the `networkView`. It is also through tag attributes that you connect auxiliary faces components to enhance the `networkView`, like the `dataSource`, `overview`, `selectInteractor`, and others

Server-side configuration can be set through a CSS configuration file or through the `IlpNetwork` API, the easiest and preferred way being the CSS configuration. The tag attribute `styleSheets` is used to pass a list of Cascading Style Sheets (CSS) files to configure the network adapter (filters, node and link factories, for example), the network view (background and zoom policies, for example) and the network objects themselves.

The following table lists all the tag attributes defined for the `networkView` component.

Tag Attributes of the `networkView` Faces Component

Tag Attributes	Description
`id`	Defines the unique identifier for the component. Every component should have a unique identifier. **Mandatory**.
`context`	Defines the JViews TGO context to be used by the underlying `IlpNetwork` component. **Mandatory**.
`binding`	Allows the user to bind the component to a backing bean.
`width`	Defines the width, in pixels, of the view component. This attribute is inherited from the `view` faces component.
`height`	Defines the height, in pixels, of the view component. This attribute is inherited from the `view` faces component.
`style`	Provides CSS customization. This attribute is inherited from the JViews Framework faces.
`styleClass`	Defines the style classes for the component. This attribute is inherited from the JViews Framework faces.
`messageBox`	Binds the text output of the `networkView` component to a `messageBox` component. (A `messageBox` component is defined by the core JViews

Tag Attributes	Description
	faces library and displays text messages in a JSP™ page.) This attribute is inherited from the JViews Framework faces.
`messageBoxId`	Similar to `messageBox` but binds the `messageBox` component by its unique identifier. This attribute is inherited from the JViews Framework faces.
`interactor`	Specifies the default (initial) interactor set to the `networkView` component. It has no correspondance with a view interactor. It should be a faces component (like the `selectInteractor` component). This attribute is inherited from the JViews Framework faces.
`interactorId`	Similar to `interactor` but binds to the unique identifier of the interactor. This attribute is inherited from the JViews Framework faces.
`zoomFactor`	Configures the zoom factor applied when zooming in or out. This attribute is inherited from the JViews Framework faces.
`panFactor`	Configures the pan factor (how much the image is moved). It is mainly used when the `networkView` component is connected with the `panTool` component. This attribute is inherited from the JViews Framework faces.
`updateInterval`	Defines the interval between automatic updates. The `networkView` component will then send an update request to the server on a regular basis. This attribute is inherited from the JViews Framework faces.
`imageFormat`	The image encoding format, for example "JPG" or "PNG". This attribute is inherited from the JViews Framework faces.
`waitingImage`	Defines the image to be displayed by the network faces component while waiting for the image servlet to generate the response image. This attribute is inherited from the JViews Framework faces.
`generateImageMap`	Indicates whether an image map should be generated for the network faces component. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapVisible`	Indicates whether the image map should be made visible or not. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapGenerator`	Binds to a bean that subclasses the `ilog.views.servlet.IlvImageMapAreaGenerator` class and is responsible for generating the image map. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapGeneratorClass`	The name of a bean class that subclasses the `ilog.views.servlet.IlvImageMapAreaGenerator` class and is responsible for generating the image map. This attribute is inherited from the JViews Framework

Tag Attributes	Description
	faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`backgroundColor`	Specifies the background color to be displayed by the network faces component when there is no background image. This attribute is inherited from the JViews Framework faces.
`onImageLoaded`	Specifies the JavaScript code to be executed right after a refreshed image is loaded from the server. This attribute is inherited from the JViews Framework faces.
`onCapabilitiesLoaded`	Similar to `onImageLoaded`, but this is executed right after a request for capabilities has been answered by the server. This attribute is inherited from the JViews Framework faces.
`errorMessage`	The error message to be displayed in case of faulty client-server communication. This attribute is inherited from the JViews Framework faces.
`servlet`	Overrides the default image servlet used to generate images. This attribute is inherited from the JViews Framework faces.
`project`	Sets a JViews TGO project to the underlying `IlpNetwork` component.
`dataSource`	Binds the `networkView` component with a `dataSource` component. The data source component wraps an `IlpAbstractDataSource` component internally.
`dataSourceId`	Similar to `dataSource`, but binding is done through the data source's unique identifier.
`network`	Specifies a custom `IlpNetwork` which replaces the default automatically instantiated network component.
`styleSheets`	Specifies a JViews TGO CSS configuration file for the underlying `IlpNetwork` component. It is different from the `styles` tag attribute as it provides server-side configuration, which is specific to JViews TGO. This CSS file may contain component and business data configuration.
`resizable`	Enables or disables the resizing of the network faces component. This attribute is inherited from the JViews Framework faces.
`boundingBox`	Defines the width and height of the network view component. This attribute is inherited from the JViews Framework faces.
`data`	Defines the data to be displayed, which can be a JViews TGO project, a binding to an `IlpAbstractDataSource` instance, or the unique identifier of a `dataSource` component.
`zoomLevels`	Comma-separated list of fixed zoom levels used by the view. If the zoom levels are not specified, the zoom is bound only by the `maxZoomLevel` property.
maxZoomLevel	The maximum zoom level. This property is used if, and only if, the `zoomLevels` property is not used.

Tag Attributes	Description
	The default value is 10.
tileSize	The size of a tile. If the tile size is greater than or equal to 0, the view will be set in tiled mode. The tile size must be carefully chosen for performance reasons.
tileManager	The tile manager is responsible for retrieving and/or storing image tiles on the server side. The tile manager is used when the view is tiled, that is, if `tileSize` is strictly positive.

Connecting a business data source

To be able to display network objects, the network faces component must be connected to a data source. This can be done in different ways:

- using a JViews TGO project

  (See *How to set a JViews TGO project to a networkView faces component*)

- using the `dataSource` faces component

  (See *How to declare a dataSource faces component for the network view* and *How to connect the dataSource faces component to the networkView faces component*)

- directly setting an `IlpAbstractDataSource`

  (See *How to set a data source Bean to a networkView faces component*)

The easiest way to provide server-side customization and business data to a network faces component is through the `project` tag attribute. It allows you to specify a JViews TGO project that will be set to the underlying `IlpNetwork` on the server side. For more information, see Loading a project file. Keep in mind that not all CSS view customizations are supported by the network faces component. For details, see *Network view component services*.

How to set a JViews TGO project to a networkView faces component

The following example shows how to pass a JViews TGO project to the `networkView` component, and to configure the component dimensions (`width` and `height`) using the `style` tag attribute:

```
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    project="data/myProject.itpr" />
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example2.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The `id` tag attribute defines a unique identifier for the `networkView` component. The `context` tag attribute is a binding to a bean defined in the `faces_config.xml` file. The `style` tag attribute defines two CSS properties (`width` and `height`) for the dimensions, in pixels, of the network component. The `project` tag attribute is a relative path to a JViews TGO project within the web application. This file should be accessible by the web application.

The following example shows how to declare the `context` bean in the `faces_config.xml` file:

```
<managed-bean>
    <managed-bean-name>contextBean</managed-bean-name>
    <managed-bean-class>
     ilog.tgo.faces.service.IltFacesDefaultContext
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The context should implement the `IlpContext` interface and must use the synchronization strategy `IlSynchronizeOnLockStrategy` in order to address the particular threading issues of a web server.

How to declare a dataSource faces component for the network view

Another way to connect business data to the network view is through the `dataSource` faces component. This component represents a wrapper for an `IlpAbstractDataSource` object that can be connected to a network component. Many different data source components can be declared in a given JSP™ page, but only one can be connected to the network view at a time. It is possible to switch data sources dynamically.

The following example shows how to declare a data source in a JSP page:

```
<jvtf:dataSource id="myDataSource" value="#{dataSourceBean}" />
```

The `id` tag attribute defines a unique identifier for the data source component. The `value` tag attribute gets a value binding to a bean previously declared in the `faces_config.xml` file that extends `IlpAbstractDataSource`.

How to connect the dataSource faces component to the networkView faces component

Once the data source has been declared, you can connect it to the network view as follows:

```
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    dataSourceId="myDataSource" />
```

The `dataSourceId` tag attribute gets the unique identifier of the data source component that will connect it to the network view.

The following example shows how to declare the `dataSource` bean in the `faces_config.xml` file:

```
<managed-bean>
  <managed-bean-name>dataSourceBean</managed-bean-name>
  <managed-bean-class>ilog.cpl.datasource.IlpDefaultDataSource</managed-bean
-class>
```

```
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>context</property-name>
    <property-class>ilog.cpl.service.IlpContext</property-class>
    <value>#{contextBean}</value>
  </managed-property>
  <managed-property>
    <property-name>fileName</property-name>
    <property-class>java.lang.String</property-class>
    <value>data/myNetwork.xml</value>
  </managed-property>
</managed-bean>
```

The `dataSource` bean is declared and two properties are set: `context` and `fileName`. The `context` property is set with a value binding to a context bean. It is mandatory, so that the JViews TGO context is consistent across components. The `fileName` property gets a relative path to an XML file compatible with the data source and accessible from the web application.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example3.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to set a data source Bean to a networkView faces component

It is also possible to set a data source bean directly to the network view component, without requiring the data source component.

For example:

```
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    dataSource="#{dataSourceBean}" />
```

The `dataSource` tag attribute gets a value binding to a bean that extends `IlpAbstractDataSource`. It will connect the network component to this data source bean.

How to use the data tag attribute of the networkView faces component

The network view faces component has a multipurpose `data` tag attribute, which can be used to connect business data sources using:

- a JViews TGO XML project file
- the unique identifier of a data source faces component
- the binding to an instance of `IlpAbstractDataSource`

Note: You must not use any combination of the following tag attributes, which allow you to connect the network view to any form of data source:

- `data`
- `dataSourceId`
- `dataSource`
- `project`

When used with JViews TGO projects, the `data` tag attribute behaves exactly like the `project` attribute, getting the relative path to a JViews TGO project, as in the following example:

```
<jvtf:networkView id="aNetwork"
                  context="#{myContext}"
                  style="width:740;height:550"
                  data="data/myProject.itpr" />
```

Here `myProject.itpr` is the project file within the web application.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example4.jsp*.

When used with the unique identifier of a data source faces component, the `data` tag attribute behaves exactly like the `dataSourceId` attribute, getting the unique identifier of a data source component, as in the following example:

```
<jvtf:networkView id="aNetwork"
                  context="#{myContext}"
                  style="width:740;height:550"
                  data="myDataSource" />
```

Here `myDataSource` uniquely identifies a data source faces component in the current session.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example5.jsp* .

When used with an `IlpAbstractDataSource` instance, the `data` tag attribute behaves exactly like the `dataSource` attribute, getting a value binding to a bean that extends `IlpAbstractDataSource`, as in the following example:

```
<jvtf:networkView id="aNetwork"
                  context="#{myContext}"
                  style="width:740;height:550"
                  data="#{dataSourceBean}" />
```

Here `#{dataSourceBean}` is a value binding to the corresponding bean declared in the `faces_config.xml` file.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example6.jsp*.

Customizing the underlying IlpNetwork component

The network view faces component is a facade to an `IlpNetwork` component which manages the integration between business data and display data, server-side configuration and interactions. By default, an instance of `IlpNetwork` is instantiated. However, you can customize the way this underlying component is created, in two different ways:

- *Using the binding tag attribute*
- *Using the network tag attribute*

Using the binding tag attribute

The `binding` tag attribute allows you to replace the default network view faces component with a customized backing bean that controls how the `IlpNetwork` is created through the `createNetworkComponent` method, as illustrated below:

```
protected IlpNetwork createNetworkComponent(IlpContext context,
                                            String config) {
  IlpNetwork myNetwork = new IlpNetwork(config, context);
  IlpDefaultDataSource dataSource = new IlpDefaultDataSource(context);
  try {
    myNetwork.setStyleSheets(new String[] { "myStyles.css" });
    dataSource.parse("myData.xml");
  } catch(Exception x) {
    System.err.println("Could not configure custom component");
  }

  myNetwork.setDataSource(dataSource);

  return myNetwork;
}
```

How to use the binding tag attribute of the networkView faces component

Faces components allow you to set a backing bean to replace the default component implementation. So, for the network view faces component, the `binding` attribute can be set with a value binding to a backing bean that extends `IltFacesDHTMLNetworkView` (the DHTML implementation of the network view faces component). The following example illustrates this:

```
<jvtf:networkView id="aNetwork"
                  context="#{contextBean}"
                  style="width:740;height:550"
                  binding="#{myJSFNetwork}" />
```

Here `#{myJSFNetwork}` is a value binding to a backing bean declared in the `faces_config.xml` like this:

```
<managed-bean>
  <description>A bean extending IltFacesDHTMLNetworkView</description>
  <managed-bean-name>myJSFNetwork</managed-bean-name>
  <managed-bean-class>example.MyNetworkView</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The backing bean provides more flexibility to the user by giving access to the component API and its instantiation.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example7.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Using the network tag attribute

The `network` tag attribute allows you to define the customized instance of an `IlpNetwork` component through method binding. You should declare a bean with a method that returns your instance of `IlpNetwork` and bind this method with the `network` tag attribute, as illustrated below:

```
public IlpNetwork getCustomNetwork() {
  if (null == network) {
    // Get the default configuration file name
    String config = IltFacesNetworkView.DefaultConfigurationFileName;
    network = new IlpNetwork(config, context);
    IlpDefaultDataSource dataSource = new IlpDefaultDataSource(context);
    try {
      network.setStyleSheets(new String[] { "myStyles.css" });
      dataSource.parse("myData.xml");
    } catch(Exception x) {
      System.err.println("Could not configure custom component");
    }

    network.setDataSource(dataSource);
  }
  return network;
}
```

Note: The configuration file is mandatory. The sample uses the default faces configuration file which is accessible from the property `IltFacesNetworkView.DefaultConfigurationFileName`.

How to use the network tag attribute of the networkView faces component

It is possible to replace the automatically created `IlpNetwork` object with a customized network object. This is done with the `network` attribute of the network view faces component, as follows:

```
<jvtf:networkView id="aNetwork"
                  context="#{contextBean}"
                  width="740"
                  height="550"
                  network="#{myIlpNetwork.network}" />
```

Here the tag attributes `width` and `height` are used to specify the size of the network view. Other examples produce the same results using the `style` tag attribute with the CSS properties "`width`" and "`height`".

In this example, the `network` attribute is set with a method that binds to a bean defined in the `faces_config.xml`. The corresponding method (`getNetwork` in this case) will be invoked when the JSP™ page is parsed. It allows the user to have access to the `IlpNetwork` API as well as to its instantiation. Using the `network` attribute and keeping the `IlpNetwork` in a bean is a good way to provide quick access to the underlying `IlpNetwork` API within the web application. Note that the context is not passed to the `myIlpNetwork.getNetwork` method, which means that this bean must be configured with the appropriate context in the `faces_config.xml` file. For example:

```
<managed-bean>
  <description>A bean with read access to the 'network' property</description>

  <managed-bean-name>myIlpNetwork</managed-bean-name>
  <managed-bean-class>example.MyNetwork</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>context</property-name>
    <property-class>ilog.cpl.service.IlpContext</property-class>
    <value>#{contextBean}</value>
  </managed-property>
</managed-bean>
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example8.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Combining faces components

You can connect components from the core JViews Faces and JViews Framework Faces libraries to the network view to combine features and improve user interaction. This is the case with the `overview`, `zoomTool`, `panTool` and `imageButton` components.

How to set up an overview for the network view

The `overview` component must be manually set up within the HTML page. Its dimensions and location are important criteria to be considered when designing the HTML page. The following example shows how to declare an overview and connect it to the network view:

```
<h:panelGrid columns="2">
  <jvtf:networkView id="aNetwork"
                       context="#{contextBean}"
                       style="width:740;height:550"
                       project="data/myProject.itpr" />
  <jvf:overview id="anOverview"
                viewId="aNetwork"
                style="width:123;height:91" />
</h:panelGrid>
```

In the example, a network view component is declared with the unique identifier "`aNetwork`" within a two-column `panelGrid`. Then, an overview component is declared so that it is layered after the network component. The `viewId` tag attribute is used to connect the network view to the overview, through the unique identifier of the main view component. Note that the dimensions of both components are defined in a similar way by the tag attribute `style`.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example9.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to connect a zoom tool and a pan tool to a network view

See section The JViews Framework Faces Component Set in the Advanced Features of JViews Framework part of the JViews Diagrammer documentation for details about the zoomTool and panTool.

The following example shows how to attach `zoomTool` and `panTool` components to a network view:

```
<h:panelGrid columns="2">
  <jvtf:networkView id="aNetwork"
                       context="#{contextBean}"
                       style="width:740;height:550"
                       project="data/myProject.itpr" />
```

```
  <h:panelGrid columns="1">
    <jvf:panTool id="aPanTool"
                 viewId="aNetwork"
                 style="width:123;height:123" />
    <jvf:zoomTool id="aZoomTool"
                  viewId="aNetwork"
                  style="width:123;height:322" />
  </h:panelGrid>
</h:panelGrid>
```

In this example, a network view component is declared with the unique identifier "`aNetwork`" within a two-column `panelGrid`. Then, a new one-column `panelGrid` is declared to accommodate the `panTool` and `zoomTool` components. The `viewId` tag attribute is used to connect the network view to the other components. Note that the `style` tag attribute is used to set the dimensions for all the declared components.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example10.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to add image buttons and set client-side actions for the network view component

Although `zoomTool` and `panTool` components provide basic user interaction, you can also set client actions to image buttons to achieve similar results. The advantage is that image buttons are more customizable, as the user can define the action to be set. The following example shows how to declare image buttons and associate them with client-side actions.

```
<!-- Create a 2 columns grid -->
<h:panelGrid columns="2">

  <!-- Declare a button for zooming in -->
  <jv:imageButton onclick="aNetwork.zoomIn(true)"
                  image="images/zoom.gif"
                  rolloverImage="images/zoomh.gif"
                  selectedImage="images/zoomd.gif"
                  title="Zoom In"
                  message="Zoom In" />

  <!-- Declare a button for zooming out -->
  <jv:imageButton onclick="aNetwork.zoomOut(true)"
                  image="images/unzoom.gif"
                  rolloverImage="images/unzoomh.gif"
                  selectedImage="images/unzoomd.gif"
                  title="Zoom Out"
                  message="Zoom Out" />
</h:panelGrid>
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
```

```
                        style="width:740;height:550"
                        project="data/myProject.itpr" />
```

This example declares two image buttons:

- one for zooming in
- one for zooming out

Each button declaration defines the following attributes:

- `onclick`: The JavaScript™ action to be triggered when the button is pressed.
- `image`: The main button image.
- `rolloverImage`: The image to be displayed when the mouse pointer rolls over the button.
- `selectedImage`: The image to be displayed when the button is pressed.
- `title`: The tooltip message displayed when the mouse pointer stays over the button.
- `message`: The message displayed in the `messageBox` component when the mouse pointer stays over the button.

The `onclick` tag attribute is the most important as it defines the action associated with the button. Note that it uses the JavaScript API of the `networkView` component to perform the desired action:

- `onclick="aNetwork.zoomIn(true)"`: This uses the `zoomIn` JavaScript call to zoom in the network view component.
- `onclick="aNetwork.zoomOut(true)"`: This uses the `zoomOut` JavaScript call to zoom out the network view component.

The `onclick` attribute can be set with any valid JavaScript code, which will be executed when the button is pressed. The other tag attributes define the look and feel of the button, with corresponding images and tooltip text.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example11.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Interacting with the network view component

JViews Framework Faces and core JViews Faces libraries declare predefined interactors that can be connected to the `networkView` component to add extra user interaction. Interactors are faces components that execute client- or server-side actions. Most of them can be extended and configured to suit the user needs.

How to declare an interactor and connect it to the network view component

The following example shows how to declare a predefined interactor (the `pan` interactor) in the JSP™ page and connect it to the `networkView` component so that it is always available.

```
<!-- Declare the predefined 'pan' interactor -->
<jvf:panInteractor id="pan" />

<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    interactorId="pan"
                    project="data/myProject.itpr" />
```

In this example, the predefined `panInteractor` is declared. A unique identifier is associated with it ("`pan`"). Then, the `interactorId` tag attribute of the `networkView` component specifies the interactor to be connected to the network view.

How to associate interactors with image buttons in the network view component

Usually many interactors are made available in a web application. The following example shows how to declare multiple predefined interactors and how to use image buttons to make them active. Note that only one interactor can be set in the network view component at a time. Whenever a new interactor is set, the previous one is removed.

```
<!-- Declare the predefined 'select' interactor -->
<jvtf:selectInteractor id="select" />

<!-- Declare the predefined 'pan' interactor -->
<jvf:panInteractor id="pan" />

<!-- Create a 4 columns grid -->
<h:panelGrid columns="4">

  <!-- Declare a button for selection -->
  <jv:imageButton onclick="aNetwork.setInteractor(select)"
                  buttonGroupId="interactors"
                  image="images/arrow.gif"
                  rolloverImage="images/arrowh.gif"
```

```
                    selectedImage="images/arrowd.gif"
                    title="Select Interactor"
                    message="Select Interactor" />

  <!-- Declare a button for panning -->
  <jv:imageButton onclick="aNetwork.setInteractor(pan)"
                    buttonGroupId="interactors"
                    selected="true"
                    image="images/pan.gif"
                    rolloverImage="images/panh.gif"
                    selectedImage="images/pand.gif"
                    title="Pan Interactor"
                    message="Pan Interactor" />

  <!-- Declare a button for zooming in -->
  <jv:imageButton onclick="aNetwork.zoomIn(true)"
                    image="images/zoom.gif"
                    rolloverImage="images/zoomh.gif"
                    selectedImage="images/zoomd.gif"
                    title="Zoom In"
                    message="Zoom In" />

  <!-- Declare a button for zooming out -->
  <jv:imageButton onclick="aNetwork.zoomOut(true)"
                    image="images/unzoom.gif"
                    rolloverImage="images/unzoomh.gif"
                    selectedImage="images/unzoomd.gif"
                    title="Zoom Out"
                    message="Zoom Out" />
</h:panelGrid>
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    interactorId="pan"
                    project="data/myProject.itpr" />
```

This example defines two predefined interactors:

- `selectInteractor`: This is a server-side interactor that processes object selection by default. (See *The selectInteractor faces component* for details).
- `panInteractor`: This is a client-side interactor that enables panning of the image displayed by the network view component.

Two buttons are declared to connect the interactor to the network view component. The `buttonGroupId` tag attribute is used to group image buttons so that only one button of the group is selected at a time. The `selected` attribute is used to specify which button should be made selected when the page is loaded. This should correspond to the interactor initially connected to the network view with the `interactorId` tag attribute. In this case, the pan button is selected (`select="true"`) and the `pan` interactor is connected to the network view (`interactorId="pan"`).

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example13.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The selectInteractor faces component

The `selectInteractor` faces component has been defined as an interactor that maps client-side mouse clicks to server-side events dispatched to the underlying view interactor. It extends the JavaServer™ Faces `UICommand` component, which means that it will fire `ActionEvents` to registered `ActionListeners`.

This component allows you to create customized `IlvManagerViewInteractor` instances that will process the mouse actions on the client side. By default, it uses the `IltSelectInteractor`, which allows selecting, dragging and expanding graphic objects.

This interactor has the following limitations in terms of handling events:

- Only `BUTTON1` and `BUTTON3` mouse buttons are supported (left and middle).
- Pop-up menus are supported in a different way from the Swing network component. See *Adding pop-up menus*.
- Double-click events are not supported.
- There is no visual feedback when dragging an object (the graphic representation of the object does not follow the mouse pointer).
- No keyboard actions are supported except the following modifiers:
  - `Shift` key
  - `Control` key
  - `Alt` key
  - `Meta` key

Client-side interactions are converted into the following mouse events, dispatched to the appropriate interactor:

- `MOUSE_PRESSED`
- `MOUSE_DRAGGED`
- `MOUSE_RELEASED`

How to declare the selectinteractor faces component for the network view

The `selectInteractor` is declared like any other faces component defined in the JViews TGO faces tag library:

```
<jvtf:selectInteractor id="select" />
```

Configuring the selectInteractor

The `selectInteractor` faces component is configured through the following tag attributes.

Tag Attributes of the `selectInteractor` Faces Component

Tag Attributes	Description
`cursor`	Defines the mouse cursor to be used when the interactor is active; it should be one of the cursors supported by the web browser.
`lineWidth`	Defines the width of the interaction area drawn by the interactor.
`lineColor`	Defines the color of the interaction area drawn by the interactor.
`actionName`	Defines the name of the action event triggered by this interactor; it is used to identify events coming from this interactor.
`autoSubmit`	Defines whether the request will be submitted by a mouse click or not; this tag is used to control when the actions are submitted. When this tag attribute is set to `false`, the `selectInteractor` will not submit any request after the user interaction, but wait until some other component does it.
`actionListener`	Defines an action listener that is called when this interactor is used. Action listeners should implement the `ActionListener` interface (from JSF library), but JViews TGO faces provide the `IltFacesGraphInteractorActionListener` abstract implementation that decodes the user interactions into events dispatched to a given view interactor. Subclasses should implement the method `getViewInteractor` in order to return the appropriate view interactor to process the events. The default implementation (`IltFacesSelectInteractorListener`) dispatches all events to the `IltSelectInteractor` view interactor. Therefore, the `actionListener` tag attribute may be used to register any `ActionListener` that will be notified whenever a user interaction has been performed, or a subclass of `IltFacesGraphInteractorActionListener` can be registered to decode the user interaction into events dispatched to view interactions (`IlpViewInteractor`).
`invocationContext`	Defines whether the server-side processing is performed in the JSF lifecycle or directly by the image servlet. The possible values are: ♦ JSF_CONTEXT: Processing is done in the JSF lifecycle (the default value) ♦ IMAGE_SERVLET_CONTEXT: Processing is done by the image servlet, bypassing the JSF lifecycle The `selectInteractor` submits requests to be processed on the server side. By default, the request is addressed to the JavaServer Faces controller servlet which processes all requests according to the well-defined JSF lifecycle. This means that all component dependencies will be verified, any registered listener will be notified. The result is a full page refresh with an update of all components involved. If your interaction triggers updates of components other than the `networkView`, then the `JSF_CONTEXT` should be used.

Tag Attributes	Description
	If, on the other hand, your requests are supposed to only update the image displayed by the `networkView` component, then you may want to benefit from the faster `IMAGE_SERVLET_CONTEXT`. In this case, the interactor requests will be addressed to the image servlet responsible for generating the image displayed by the `networkView` component. Note that the image servlet has no access to any faces component other than `networkView`, which means that your request cannot rely on other faces components, and that any registered listener for changes on the `selectInteractor` and `networkView` will not be updated. There is one exception to this rule. As the `overview` faces component is largely used with the `networkView` faces component, and as its displayed image is also generated by the image servlet, you can force the overview to be refreshed whenever the main view (`networkView`) is updated, at the cost of an extra client-server-client roundtrip. To do so, you must set the `autoRefresh` tag attribute of the `overview` faces component to true.
`binding`	Defines a binding expression to a backing bean.

The `cursor`, `lineWidth` and `lineColor` tag attributes control the look of the interactor when it is activated, they do not affect its functionality.

If you have started the bundled Tomcat web server, the following links will take you to the small samples illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example12.jsp* and *http://localhost:8080/jsf-network-step-by-step/faces/example13.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring action listeners for the select interactor component

Action listeners are responsible for processing the interactions performed with the select interactor component. Their default behavior is to convert the interactions into server events dispatched to the `IltSelectInteractor`. It is possible to override this behavior by adding action listeners to the component.

Unlike in the regular network Swing component, it is not possible to declare view interactors through a CSS file. Instead, in the network faces component, the view interactors are declared within customized action listeners added to the `selectInteractor` faces component.

As the `selectInteractor` extends the JavaServer Faces `UICommand`, it allows one or more action listeners (implementing `javax.faces.event.ActionListener`) to be registered to receive events (`javax.faces.event.ActionEvent`) whenever a user interaction is performed. For the `ActionEvent` API there are the following methods:

- `getComponent()` or `getSource()`: Return a reference to the interactor faces component that is currently active (for example, `IltFacesGraphInteractor`).

A predefined abstract implementation of the `ActionListener` interface named `IltFacesGraphInteractorActionListener` is provided to translate client-side interactions into server-side events that are dispatched to a given view interactor. When notified, this class translates user interactions into mouse events that are automatically dispatched to the `IlpViewInteractor` returned by the abstract method `getViewInteractor(actionName)`.

The following example illustrates how to override the default `selectInteractor` behavior with a customized one:

```
<jvtf:selectInteractor id="select"
                       actionListener="#{MyListenerBean}"
                       invocationContext="JSF_CONTEXT" />
```

Here the `actionListener` tag attribute gets a binding to a bean implementing the `javax.faces.event.ActionListener` interface. Note that `actionListener` will override the default behavior of the `selectInteractor`. It is possible to add more than one action listener, combining customized action listeners with the default behavior as shown in the next example:

```
<jvtf:selectInteractor id="select"
                       invocationContext="JSF_CONTEXT">
  <f:actionListener
   type="ilog.tgo.faces.graph.dhtml.event.IltFacesSelectInteractorListener"/
>
   <f:actionListener type="demo.MyInteractionListener"/>
</jvtf:selectInteractor>
```

Here `IltFacesSelectInteractorListener` is the action listener (extends `IltFacesGraphInteractorActionListener`) that implements the default behavior of the `selectInteractor` faces component, and `MyInteractionListener` is a customized implementation of the `javax.faces.event.ActionListener` interface. The `actionListener` tag is used to add several action listeners to the `selectInteractor`, which are invoked in the order in which they have been declared. Note that action listeners may conflict with each other, especially multiple implementations of `IltFacesGraphInteractorActionListener`, as the first one invoked may change the business model and invalidate the next action listener.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to customize action listeners: *http://localhost:8080/jsf-network-step-by-step/faces/example14.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The clientSelectInteractor faces component

The `clientSelectInteractor` faces component is an interactor designed to minimize the number of image requests and image updates between the graph view on the client and the image servlet on the server by dynamically rendering and managing the selection borders in the client side.

Instead of requesting a new graph view image every time the user selects an object, the `clientSelectInteractor` dynamically renders an HTML rectangular selection around the object. The server is notified so that the selection model is kept synchronized with the user interactions. A new graph view image is requested only when the user drags objects or interacts with specific decorations (such as information icons, and expansion icons).

The performance and responsiveness is greatly improved as lesser images are generated and dispatched by the server. However, the selection graphic feedback is impacted, as the client is limited to only displaying a rectangular border around the selected object.

The interactor can be configured to work in *image* mode. In this mode, it will ask the server to process the selection and get a new image on every user interaction. The dynamic selection border will only be displayed while objects are being dragged, as when objects are in their resting position the new image sent by the server already represents the selection.

It is possible to customize the interaction with object decorations, controlling when a click on a particular object decoration should trigger a new image request or not.

Note: Unlike the `selectInteractor`, the `clientSelectInteractor` always communicates with the image servlet directly. Therefore, it does not follow the JSF lifecycle, which means that only the view faces component and a possibly attached overview are updated and not all the other components in the page.

How to declare the clientSelectInteractor faces component for the network view

The `clientSelectInteractor` is declared like any other faces component defined in the JViews TGO faces library:

```
<jvtf:clientSelectInteractor is="clientSelect" />
```

Configuring the clientSelectInteractor

The `clientSelectInteractor` faces component is configured through the following tag attributes:

Tag Attributes of the `clientSelectInteractor` Faces Component

Tag Attributes	Description
`binding`	Defines a binding expression to a backing bean, allowing the user to customize the `clientSelectInteractor` faces component.
`message`	Defines the message displayed by the view when the interactor is set.
`cursor`	Defines the mouse cursor to be used when the interactor is active. It should be one of the cursors supported by the web browser.
`menuModeId`	The identifier passed to dynamically generated popup menus.
`moveAllowed`	Specifies whether this interactor allows mouse dragging.
`objectActionMethodBinding`	Defines a method binding to process actions on the object or on specific decorations attached to it.
`onSelectionChanged`	Specifies a JavaScript handler to be called whenever the selection changes. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`imageMode`	Sets the interactor to image mode. A new image will be requested every time the user interacts with the view. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`lineWidth`	The width of the selection border dynamically rendered by the interactor. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`lineColor`	The color of the selection border dynamically rendered by the interactor. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`forceUpdateProperties`	Forces the request of additional information when the interactor is in image mode. Used in conjunction with the `infoProviderMethodBinding`. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`infoProviderMethodBinding`	Defines the information provider method binding to return additional information about the selected object. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.

If you have started the bundled Tomcat web server, the following link will take you to a small sample illustrating this: *http://localhost:8080/jsf-network-step-by-step/faces/example19.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring an object action for the clientSelectInteractor

By default, the `clientSelectInteractor` supports interactions with the following object decorations:

- The information icon

- The system icon
- The expand and collapse icons

If the user clicks on any of these decorations, the interactor triggers a default object action instead of selecting the object. These actions change the look of the objects, which means that a new image is generated.

It is possible to override or even extend this behavior through the `objectActionMethodBinding` tag attribute, as follows:

```
<jvtf:clientSelectInteractor id="clientSelect"

objectActionMethodBinding="#{interactorBean.objectAction}" />
```

Here, the `objectActionMethodBinding` tag attribute is bound to the `objectAction` method declared in the `interactorBean`. The method binding must conform to the following signature:

```
boolean methodName(IlpGraphView, int, int)
```

The `IlpGraphView` is a reference to the graph view containing all the objects. The two integer parameters are the `x` and `y` components of the view position where the user has clicked. Returning `true` indicates that a new image has to be generated, while `false` indicates that nothing has been processed and the clicked object will be selected.

In the example above, the `objectAction` method will override the default object action behavior (which is to react on expand, collapse, information and system icons). It is possible to extend this default behavior by overriding the method `processObjectAction` in class `IltFacesDefaultObjectAction`.

The selectionManager faces component

You can customize the way the selection is performed and displayed by using a selection manager.

This selection manager is defined in a facet on the `networkView` tag, as follows:

```
<jvtf:networkView id="tgoViewId" interactorId="select">
  <f:facet name="selectionManager">
      <jvtf:selectionManager imageMode="false" [...] />
  </f:facet>
</jvtf: networkView >
```

The selection manager has two display modes:

- image (default)

  The image is refreshed after each selection. A new image is requested to the server at each selection which allows the client to get nice selection graphics.

- regular

Rectangles representing the selection are displayed on top of the view. The roundtrip to the server is minimal: the generation of a new image is not required and the response time is faster but the selection feedback is limited to a selection rectangle.

Tip: Image Mode versus Regular Mode

Using one mode rather than the other depends on your criteria: performance or graphic feedback. Image mode provides a better graphic feedback but is slower because of the image generation and the need for an extra request to get additional information about the selection on the client. Regular mode offers basic graphic feedback but better performance.

Other parameters can be configured on the `selectionManager`, like for example the line width or the color of the selection rectangle used in regular selection mode:

```
<jvtf:selectionManager lineWidth="2" lineColor="red"/>
```

Note: The selection manager currently supports integration with the following TGO Faces interactor: `clientSelectInteractor`.

Configuring the selectionManager

The `selectionManager` faces component is configured through the following tag attributes:

Tag Attributes of the `selectionManager` Faces Component

Tag Attributes	Description
`infoProviderMethodBinding`	A method binding that respects the signature List methodName(IlpGraphView, IlpRepresentationObject). The returned value of this method is a list of additional properties associated with the selected object. A valid item of this list is a String or a list itself. As of JViews TGO 8.0, the preferred way to transfer object properties to client is via the `propertyAccessor` tag of the `selectionManager`.
`binding`	The value binding expression linking this component to a property in a backing bean. If this attribute is set, the tag does not create the

Tag Attributes	Description
	component itself but retrieves it from the Bean property. This attribute must be a value binding.
`lineColor`	The color of selection rectangles lines.
`forceUpdateProperties`	Force to make additional request to query the current selection and additional properties in image mode to enable client-side selection listener.
`id`	The ID of this component.
`imageMode`	The image mode. In image mode, the image is refreshed on each selection. In regular mode, only the selected object(s) bounding box is queried and rectangles are dynamically displayed on top of the view. Note that the client-side listeners on selection and additional information on selected objects are available in image mode if and only if the `forceUpdateProperties` property is set to true. In regular mode no special configuration is needed. By default the manager is in image mode.
`lineWidth`	The width of selection rectangle lines.
`onSelectionChanged`	A JavaScript handler called when the selection has changed. The handler can use the predefined variable '`selection`' which is the list of current selected items. To use this handler the `selectionManager` must be in regular mode or the `forceUpdateProperties` must be set if in image mode. Refer to the user's documentation for further information.
`propertyAccessor`	The reference to the value binding expression to an `IltFacesPropertyAccessor` instance that will be used to access model properties of the selected objects.
`fillOn`	true to display filled selection rectangles. The fill color is the line color with a transparency of 50%.

Exposing selection details

You can expose details on the current selection by taking advantage of the property accessor of the selection manager.

The `IltFacesPropertyAccessor` contains several methods that can be overridden to configure or specialize the way it gives access to model properties. In particular, you can filter the properties that are exposed to clients by overriding the following method:

```
List getPropertyNames(IlpGraphicView view, IlpRepresentationObject object)
```

The following code illustrates how to provide your property accessor:

```
<jvtf:selectionManager propertyAccessor="{#serverBean.propertyAccessor}" [...
]
/>
```

The following code illustrates how to implement your custom requirements in a new property accessor:

```
public class ServerBean {

  private IltFacesPropertyAccessor accessor = new MyPropertyAccessor();

  public IltFacesPropertyAccessor getPropertyAccessor() {
    return accessor;
  }

  class MyPropertyAccessor extends IltFacesPropertyAccessor {

   protected List getPropertyNames(IlpGraphicView view, IlpRepresentationObject
 object) {
      […]
    }
  }
}
```

Then you can register a JavaScript listener that will be called when the selection changes:

```
<jvtf:selectionManager onSelectionChanged="displayProperties(selection)"/>
```

The JavaScript function can be as follows:

```
// Alert the ID and bounds of all the selected objects
function displayProperties(selection) {
  for (var i = 0; i < selection.length; i++)
   alert(selection[i].getID()+"+selection[i].getBounds());
}
```

In addition to the ID and bounds properties of the selected object, you can also expose the properties of the selected object in the TGO model as follows:

```
// Alert all the properties of all the selected objects
function displayProperties(selection) {
  for (var i = 0; i < selection.length; i++) {
    var propertiesNames = selection[i].getObjectPropertyNames();
    for (var j = 0; j < propertiesNames.length; j++)
         alert(selection[i].getObjectProperty(propertiesNames[j]));
  }
}
```

Note: To obtain selected object properties information on the client side while you are running the selection in image mode, you need to force an additional request by setting the property `forceUpdateProperties` to `true`. In regular mode this feature is available without any overhead.

If you have started the bundled Tomcat Web server, the following link will take you to the small sample illustrating how to use the selection manager with the property accessor: *http://localhost:8080/jsf-network-step-by-step/faces/example20.jsp*.

You can find more information about the sample Web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` is the directory in which IBM® ILOG® JViews TGO is installed.

Managing object selection

The `selectionManager` also allows for changing the selection state of objects programmatically via its client side by means of JavaScript. This can be accomplished by using the following API:

- `IlvAbstractSelectionManager.selectById(id, extend)`
- `IlvAbstractSelectionManager.selectAll()`
- `IlvAbstractSelectionManager.deselectAll()`

Selecting and deselecting an object

The `selectById` function allows you to select or deselect an object by providing the object's identifier:

```
networkView.getSelectionManager().selectById("tgoObjectId");
```

This method call will select the object with the identifier `tgoObjectId` and deselect the object currently selected.

You can extend/reduce the current selection by selecting/deselecting a node as follows:

```
networkView.getSelectionManager().selectById("tgoObjectId", true);
```

This method call keeps the existing selection and selects the object with the identifier `tgoObjectId`, if it's not already selected; or it will deselect it, if it is already selected.

Selecting all objects

The `selectAll` function allows you to select all objects:

```
networkView.getSelectionManager().selectAll();
```

This method call will select all visible objects.

Deselecting all objects

The `deselectAll` function allows you to deselect all objects:

```
networkView.getSelectionManager().deselectAll();
```

This method call will clear the selection of all visible objects.

Note: In all cases, the object must be selectable in order to get selected.

If you have started the bundled Tomcat Web server, the following link will take you to the small sample illustrating how to use the selection manager API: *http://localhost:8080/jsf-network-step-by-step/faces/example22.jsp*.

You can find more information about the sample Web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` is the directory in which IBM® ILOG® JViews TGO is installed.

Zoom constraints

When the zoom level is equal to 1, the manager content is adjusted to the bounds of the JSF view so as to be displayed entirely. Consequently, a zoom level of `n` means that the content is scaled by a factor of `n`. For example, a zoom factor of 2 means that the manager content is displayed double its size.

By default, the view is constrained by the manager content bounds. The direct consequences are that:

- Pan actions or zoom interactions cannot go out of the manager content bounds.
- The view zoom level cannot be lower than 1.

This constraint can be removed by setting the `constrainedOnContents` property to `false`, as follows:

```
<jvtf:networkView constrainedOnContents="false" [...] />
```

The zoom level applied to the view by using the zoom interactor of JavaScript™ zoom actions can be free or constrained to specified zoom levels. In the free zoom mode, the only constraints are the minimum and maximum zoom levels. The default value of the minimum zoom level is set to 1 and the default value of the maximum zoom level is set to 10. These constraints can be customized with the `minZoomLevel` and the `maxZoomLevel` properties respectively.

```
<jvtf:networkView minZoomLevel="2" maxZoomLevel="20" [...] />
```

Note: By default, the minimum zoom level cannot be lower than 1.

To specify fixed zoom levels, use the `zoomLevels` property, as follows:

```
<jvtf:networkView zoomLevels="1.0, 2.0, 5.0, 10.0" [...] />
```

When this property is set:

- The `minZoomLevel` and `maxZoomLevel` properties are ignored.
- The `minZoomLevel` becomes the first zoom level and the `maxZoomLevel` the last zoom level in the list.
- The zoom interactor will fit to the nearest zoom level.
- The built-in zoom actions on the JavaScript view proxy use these fixed zoom levels.

Fixed zoom levels must be used in order for a tiled view to be cached on the client-side.

For more details on setting up zooming in the network view see *How to associate interactors with image buttons in the network view component*.

Controlling the displayed area

The Network Faces component allows developers to specify the area that will be displayed on the client. For example, this enables developers to set the initial visible area or possibly to change at runtime the clipping rectangle so that it centers or focuses on a given network element.

This can be done by means of the `boundingBox` property as follows:

```
<jvtf:networkView [...] boundingBox="0,0,100,200"/>
```

The value provided corresponds to the `x`, `y`, `height` and `width` of the area of interest in manager coordinates separated by commas.

Programmatically, this property can be used during a JSF action to reset or modify the visible area by providing an instance of `IlvRect` as illustrated below.

```
public class ActionProvider {
  [...]

  public void changeAreaDisplayed() {
    IltFacesNetworkView facesNetworkView = …;
    facesNetworkView.setBoundingBox(new IlvRect(0,0,100,100));
  }
}
```

Adding pop-up menus

Unlike the network Swing component, the network faces component does not rely on the `IlpPopupMenuFactory` interface to declare contextual menus. Instead, it is based on the `contextualMenu` tag defined in the `jviews-framework-faces.tld` tag library descriptor. This means that pop-up menus in network faces cannot be declared in CSS files.

The `contextualMenu` tag allows you to define two distinct types of pop-up menu:

- Static pop-up menus: The menu structure is hard coded in the JSP™ file, it applies to all objects and cannot be changed dynamically.
- Dynamic pop-up menus: The menu structure is defined by the `IlvMenuFactory` interface and can be created dynamically where the pop-up was activated

Note: In JViews TGO Faces, the pop-up menu does not trigger any object selection, that is, the object right below the mouse pointer is not automatically included in the selection model.

How to add a static pop-up menu to a network faces component

The static pop-up menu is fully declared within the JSP file, using the following tags:

- `contextualMenu` (`jviews-framework-faces.tld` library)
- `menu` (`jviews-faces.tld` library)
- `menuItem` (`jviews-faces.tld` library)
- `menuSeparator` (`jviews-framework-faces.tld` library)

The following example illustrates how to declare a static pop-up menu within a network faces component:

```
<!-- Declare the Network Faces component -->
<jvtf:networkView id="aNetwork"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    project="data/default_project.xml">
  <!-- Declare the contextual menu -->
  <jvf:contextualMenu>
    <!-- Declare the root popup menu -->
    <jv:menu label="root">
      <jv:menuItem label="Zoom In"
                   image="images/zoom.png"
                   onclick="aNetwork.zoomIn()" />
      <jv:menuItem label="Zoom Out"
                   image="images/unzoom.png"
                   onclick="aNetwork.zoomOut()" />
```

```
        <jv:menuSeparator />
        <jv:menuItem label="Fit To Contents"
                     image="images/zoomfit.png"
                     onclick="aNetwork.showAll()" />
        <jv:menuItem label="Alert!"
                     image="images/alert.png"
                     onclick="alert('Alert menu item!')" />
      </jv:menu>
    </jvf:contextualMenu>
  </jvtf:networkView>
```

In this example, the `contextualMenu` tag is declared within the network faces component declaration (`networkView` tag). It is structured as a root menu (`menu` tag) with multiple menu items (`menuItem` tags).

The `onclick` attribute in the `menuItem` tag is the most important. It defines the JavaScript™ code to be executed when the menu item is selected. See `index` for details on the available tag attributes.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to declare a static pop-up menu: *http://localhost:8080/jsf-network-step-by-step/faces/example15.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to add and customize a dynamic pop-up menu for a network faces component

Like the static pop-up menu, the dynamic pop-up menu is declared in a JSP page using the `contextualMenu` tag inside the network faces declaration (`networkView` tag). However, instead of declaring the menu structure, it declares a menu factory (implementing the `IlvMenuFactory` interface) that is invoked whenever the pop-up menu is activated. The following exampe illustrates how the dynamic pop-up menu is declared:

```
<head>
  <!-- Specify a CSS file -->
  <link href="data/style.css" rel="stylesheet" type="text/css"/>
</head>

<!-- Declares a select interactor, which will be attached to the view -->

<jvtf:selectInteractor id="select"
                       menuModelId="selectInteractor"
                       invocationContext="IMAGE_SERVLET_CONTEXT" />

<!-- Declare the Network Faces component -->
<jvtf:networkView id="aNetwork"
                  context="#{contextBean}"
                  interactorId="select"
                  backgroundColor="#F5F5F5"
```

```
                        style="width:740;height:550"
                        project="data/default_project.xml">
      <!-- Declare the contextual menu with given popup menu factory -->
      <jvf:contextualMenu factory="#{popupMenuFactory}"
                          itemStyleClass="menuItem"
                          itemHighlightedStyleClass="menuItemHighlighted"
                          itemDisabledStyleClass="menuItemDisabled" />
    </jvtf:networkView>
```

As shown above, the `contextualMenu` tag is used within the `networkView` declaration to add a pop-up menu to the network faces component. In addition, the following tag attributes are noteworthy:

The factory tag attribute

This attribute of the `contextualMenu` tag is bound to a bean implementing the `IlvMenuFactory` interface, which defines one single method:

```
  public IlvMenu createMenu(Object graphicComponent, Object selectedObject,
String menuModelId);
```

When this method is automatically called, the `graphicComponent` attribute refers to the underlying graphic view (`IlpGraphView`, superclass of `IlpNetworkView`). It allows full access to the `IlpNetworkView` API, including selection model, controller, and so on.

The `selectedObject` attribute refers to the representation object (`IlpRepresentationObject`) located immediately below the mouse pointer when the pop-up menu was activated, if any. Note that this object may or may not be selected. It is independent of the selection model.

The `menuModelId` corresponds to the value set in the `menuModelId` tag attribute of the `selectInteractor` tag. It allows you to create custom pop-up menus based on the active interactor.

The following `IlvMenuFactory` example creates a basic pop-up menu:

```
  public IlvMenu createMenu(Object graphicComponent,
                            Object selectedObject,
                            String menuModelId) {
    // Create the root menu
    IlvMenu root = new IlvMenu("Root");

    // Create 3 JavaScript actions
    ActionListener jsAction = new
JavaScriptActionListener("aNetwork.zoomIn()");
    root.addChild(new IlvMenuItem("Zoom in", jsAction,
                                  "images/zoom.png", true));

    jsAction = new JavaScriptActionListener("aNetwork.zoomOut());
    root.addChild(new IlvMenuItem("Zoom out", jsAction,
                                  "images/unzoom.png", true));

    jsAction = new JavaScriptActionListener("alert('Alert menu item!')");
    root.addChild(new IlvMenuItem("Alert!", jsAction,
```

```
                                    "images/alert.png", true));

    return root;
  }
```

In this example, `IlvMenu` is the root menu that contains menu items (`IlvMenuItem`). Each menu item has an `ActionListener` associated with it. In this case, the predefined `JavaScriptActionListener` class is used to trigger JavaScript code executed on the client when the corresponding menu item is activated. Note that `aNetwork` in `aNetwork.zoomOut()` refers to the identifier of the `networkView` faces component. `zoomOut()` is the JavaScript method that performs zooming out on the client side.

The itemStyleClass, itemHighlightedStyleClass and itemDisabledStyleClass tag attributes

These attributes of the `contextualMenu` tag are used to customize the look of the pop-up menu. They declare the CSS classes that contain styling definitions for items, highlighted items and disabled items, respectively as follows (from the `style.css` file):

```
.menuItem {
  background: #E5E5E5;
  font-family: sans-serif;
  font-size: 14px;
  font-style: normal;
  color: black;
}

.menuItemHighlighted {
  background: #FFE5A5;
  font-style: normal;
  color: black;
}

.menuItemDisabled {
  font-style: italic;
  color: #A5A5A5;
}
```

The menuModelId tag attribute

This attribute of the `selectInteractor` tag is used by the menu factory to identify which pop-up menu to create based on the interactor that is currently active.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to customize pop-up menus: *http://localhost:8080/jsf-network-step-by-step/faces/example16.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to trigger server actions from a dynamic pop-up menu of a network faces component

The dynamic pop-up menu can trigger two types of action:

- client actions: JavaScript actions executed on the client
- server actions: Java™ actions executed on the server

When building the dynamic menu, the pop-up menu factory (`IlvMenuFactory`) creates a root menu (`IlvMenu`) with menu items (`IlvMenuItem`) and each menu item has an action listener (`ActionListener`) associated with it.

Client actions are defined by the predefined `JavaScriptActionListener`.

Server actions, like interactions, can be processed either by the JavaServer Faces lifecycle or directly by the image servlet. This is defined by an invocation context that can be either one of the following:

- `JSF_CONTEXT`: Processing takes place in the JSF lifecycle (default value)
- `IMAGE_SERVLET_CONTEXT`: Processing is performed by the image servlet, bypassing the JSF lifecycle.

Server actions are defined by subclassing the `FacesViewActionListener` abstract class. The subclass should define the desired invocation context and implement the `public void actionPerformed(EventObject event)` method. The event parameter is in fact an instance of the `ServletActionListener` class that has the following convenient methods in its API:

- `getGraphicComponent()`: This method returns the underlying view (instance of `IlpNetworkView`)
- `getObject()`: This method returns the representation object (`IlpRepresentationObject`) located right below the mouse pointer when the pop-up menu was activated)

This allows full access to the `IlpNetworkView` API, including selection model, controller, and so on.

The following example shows a basic subclass of `FacesViewActionListener`:

```
public class MyActionListener extends FacesViewActionListener {
  /**
   * Constructor. Sets the invocation context.
   */
  public AddAlarmActionListener() {
    super(IlvDHTMLConstants.IMAGE_SERVLET_CONTEXT);
  }

  /**
   * Access the network view and the active object.
   *
   * @param event An instance of ServletActionListener.
   */
  public void actionPerformed(EventObject event) throws Exception {
    ServletActionEvent saEvt = (ServletActionEvent)event;
```

```
    // access the network view
    IlpNetworkView view = (IlpNetworkView)saEvt.getGraphicComponent();

    // access the active object
    IlpObject obj = (IlpObject)saEvt.getObject();

    // implement your action with 'view' and 'obj'
  }
}
```

Once the action listener has been defined, it can be used within the pop-up menu factory (`IlvMenuFactory`) as follows:

```
public IlvMenu createMenu(Object graphicComponent, Object selectedObject,
                          String menuModelId) {
  // Create the root menu
  IlvMenu root = new IlvMenu("Root");

  // Create one server action
  ActionListener srvAction = new MyActionListener();
  root.addChild(new IlvMenuItem("My action", srvAction,
                                "images/action.png", true));

  return root;
}
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to handle server actions: *http://localhost:8080/jsf-network-step-by-step/faces/example17.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-network-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Tiling

The Network Faces component provides support for tiling. The tiling support consists of providing developers with the ability of configuring the Network Faces component to compute, cache and provide on demand only the areas of its graphical representation that are visible to the client at one given time, instead of computing and providing the entire area of the graphical representation that may not be visible to the client.

Note: The tiling support provided by the JViews TGO Network Faces component is based on the tiling support by the underlying JViews Faces Framework. Any difference in the default behavior as defined in the JViews Faces Framework documentation is documented in this section.

Configuration

Tiled view

See Concepts in Advanced Features of JViews Framework for an introduction to the use of tiling for building Web applications.

To make tiling available in the view, you must specify a tile size. The tile size is a critical parameter and must be chosen with care. See Tile Size in Advanced Features of JViews Framework.

```
<jvtf:networkView [...] tileSize="256"/>
```

Server-side caching

When the view is tiled, a server-side caching mechanism for tiles of static layers can be installed by using the `tileManager` property. No server-side caching mechanism is installed by default.

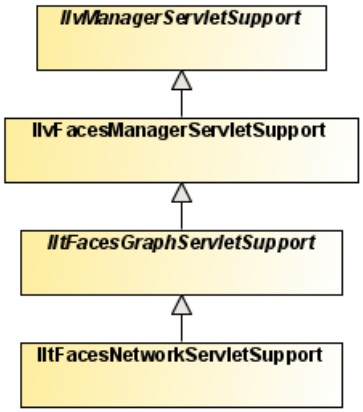
```
<jvtf:networkView tileManager="#{tgoBean.tileManager}" [...]/>
```

See *The Tile Manager* in Advanced Features of JViews Framework for more information.

Client-side caching

In order to enable the client-side caching mechanism, the `zoomLevels` attribute must be set. When this attribute is set, the client caches the tiles for the predefined zoom levels. See *Zoom constraints* for details on how to set the predefined zoom levels.

The API



IltFacesGraphServletSupport

The `IltFacesGraphServletSupport` determines the JViews TGO specific tiling behavior at the server-side level. By default, it uses as the static layers all the `IlvManagerLayer` instances that compose the background of your Network Faces component. For more information on backgrounds, see Background support.

If a custom strategy is needed for computing the tiled layers, see Developing server-side tiling which gives more information on the relevant server-side API that is need to customize the default behavior.

Managing the session expiration

The user session expires after a certain period of inactivity, usually defined in the Web deployment descriptor.

JViews objects are stored in the HTTP user session. For example, after the user session expiration, queries to update the image will fail.

The `beforeSessionExpirationHandler` property allows you to add a JavaScript™ handler that will be invoked when the user session is about to expire.

For example, to keep the session alive as long as the browser page is open, use the following code:

```
<jvtf:networkView [...]
    beforeSessionExpirationHandler="view.updateImage();" />
```

This example shows how to query an image and keep the user session alive.

Note the use of `view`, the implicit object that represents the view JavaScript proxy. The internal timer is reset only by requests issued by JViews objects. If the application implements other requests that do not refresh the image, this timer could be inaccurate. To reset the timer manually, use the following JavaScript code:

```
viewID.getObject().resetSessionExpirationTimer();
```

where `viewID` is the value of the `id` property of your view component.

Note: The `beforeSessionExpirationHandler` is called two minutes before the actual session expiration time.

Network view component services

Most of the JViews TGO network services can be used in the network faces context, with no modifications.

- Interacting with the network objects
- Positioning
- Node layout
- Link layout
- Label layout
- Layers
- Background support
- Filtering
- Accepted and excluded classes
- Setting the list of origins
- Node factory
- Link factory
- Expansion strategy

For more information on these services, refer to the Network component services.

The following services show some differences in the network faces component:

- Interacting with the network view
- Zooming

Interacting with the network view

JViews TGO network faces interactors are declared in the JSP™ file. This is required to specify how the web browser will react to user input; some interactions being executed directly on the client side while others are submitted to and executed on the server.

View interactors cannot be purely declared in the CSS file (as is the case for the Swing network component). Instead, they have to be declared together with the `selectInteractor` tag. For details on how to set a specific view interactor as the listener of a `selectInteractor`, refer to *Configuring the selectInteractor*.

Zooming

Although all three zooming modes (physical zoom, logical zoom and mixed zoom) are supported in the Faces Network component, some thresholds are affected by some of the

IBM® ILOG® JViews Network Faces specific settings, like `zoomFactor`, `zoomLevels`, `minZoomLevel`, or `maxZoomLevel`.

The equipment view faces component

Explains how to build and interact with an equipment faces component.

In this section

Declaring an equipment view faces component
Describes how to declare an equipment view faces component.

Configuring an equipment view faces component
Explains how to configure the rendering of an equipment faces component.

Equipment view component services
Presents the services that are fully compatible.

Declaring an equipment view faces component

The equipment view faces component displays the contents of an `IlpEquipment` in a JavaServer™ Page (JSP™) compliant with the JavaServer Faces (JSF) technology. It is implemented by the class `IltFacesEquipmentView` and acts as a facade to an `IlpEquipment` component. It provides a convenient API for the most common uses of the equipment component, such as setting or retrieving the associated data source, accessing the underlying equipment component, or accessing the equipment view directly.

JViews TGO faces components are declared in a tag library descriptor (`.tld`) file named `jviews-tgo-faces.tld` that is included in the `jview-tgo-all.jar`. The JViews TGO faces tag library must be declared in the JSP page before any of its components are used.

How to define the JViews TGO faces tag library and prefix in a JSP page

The declaration is done at the beginning of the JSP file as follows:

```
<%@ taglib uri="http://www.ilog.com/jviews/tlds/jviews-tgo-faces.tld"
prefix="jvtf" %>
```

This statement declares the `jviews-tgo-faces.tld` tag library within a JSP page, and binds all its components to the `jvtf` prefix. Once this is done, you can declare the equipment view component as follows:

How to declare an equipment view faces component

```
<jvtf:equipmentView id="myEquipment"
                    context="#{myContext}"
```

The `equipmentView` component requires two mandatory tag attributes:

- `id` (component unique identifier): Can be any given string that uniquely identifies this component within a server session.
- `context` (the `IlpContext` to be used): Should be a value binding to an instance of `IlpContext` declared as a managed bean. A default implementation is available for convenience (`ilog.tgo.faces.service.IltFacesDefaultContext`).

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example1.jsp*.

You will find more information about the sample web application in **<*installdir*> /samples/faces/jsf-equipment-step-by-step/index.html** where <installdir> stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring an equipment view faces component

Explains how to configure the rendering of an equipment faces component.

In this section

Configuring the client and server side of the equipmentView component
Describes the tag attributes defined for the equipmentView component.

Connecting a business data source
Explains the different ways to configure a data source within the equipment faces component.

Combining faces components
Describes how to connect components from the core JViews Faces and JViews Framework Faces libraries to the network view.

Interacting with the equipment view component
Describes how to declare predefined interactors and connect them to the equipmentView component.

Zoom constraints
Describes how to specify zoom levels.

Controlling the displayed area
Describes how to control the area displayed on the client.

Adding pop-up menus
Explains how to define pop-up menus by means of the contextualMenu tag.

Tiling
Describes the tiling support provided by the Equipment Faces component.

Managing the session expiration
Explains how to manage the user session expiration.

Configuring the client and server side of the equipmentView component

To display an equipment view in the client application, you need business data and rendering information that defines how to display these data. This section explains how to connect to a source of business data and configure their rendering in the equipment faces component.

The configuration for the rendering is split into two distinct groups:

- client-side configuration: HTML configuration stored in the DHTML page and the JavaScript™ objects
- server-side configuration: stored in the equipment faces implementation or in the image servlet

Client-side configuration relates to the behavior and look of the faces component itself. Server-side configuration relates to the equipment model and the way the representation objects are displayed and laid out.

There are many ways to configure the client and server sides of the `equipmentView` component. In general, the client-side configuration is passed as tag attributes to the `equipmentView`. It is also through tag attributes that you connect auxiliary faces components to enhance the `equipmentView`, like the `dataSource`, `overview`, `selectInteractor`, and others

Server-side configuration can be set through a CSS configuration file or through the `IlpEquipment` API, the easiest and preferred way being the CSS configuration. The tag attribute `styleSheets` is used to pass a list of Cascading Style Sheets (CSS) files to configure the equipment adapter (filters, node and link factories, for example), the equipment view (background and zoom policies, for example) and the equipment objects themselves.

The following table lists all the tag attributes defined for the `equipmentView` component.

Tag Attributes of the `equipmentView` Faces Component

Tag Attributes	Description
`id`	Defines the unique identifier for the component. Every component should have a unique identifier. **Mandatory**.
`context`	Defines the JViews TGO context to be used by the underlying `IlpEquipment` component. **Mandatory**.
`binding`	Allows the user to bind the component to a backing bean.
`width`	Defines the width, in pixels, of the view component. This attribute is inherited from the `view` faces component.
`height`	Defines the height, in pixels, of the view component. This attribute is inherited from the `view` faces component.
`style`	Provides CSS customization. This attribute is inherited from the JViews Framework faces.
`styleClass`	Defines the style classes for the component. This attribute is inherited from the JViews Framework faces.
`messageBox`	Binds the text output of the `equipmentView` component to a `messageBox` component. (A `messageBox` component is defined by the core JViews faces library and displays text messages in a JSP™ page.) This attribute is inherited from the JViews Framework faces.
`messageBoxId`	Similar to `messageBox` but binds the `messageBox` component by its unique identifier. This attribute is inherited from the JViews Framework faces.
`interactor`	Specifies the default (initial) interactor set to the `equipmentView` component. It has no correspondance with a view interactor. It should be a faces component (like the `selectInteractor` component). This attribute is inherited from the JViews Framework faces.
`interactorId`	Similar to `interactor` but binds to the unique identifier of the interactor. This attribute is inherited from the JViews Framework faces.
`zoomFactor`	Configures the zoom factor applied when zooming in or out. This attribute is inherited from the JViews Framework faces.
`panFactor`	Configures the pan factor (how much the image is moved). It is mainly used when the `equipmentView` component is connected with the

Tag Attributes	Description
	`panTool` component. This attribute is inherited from the JViews Framework faces.
`updateInterval`	Defines the interval between automatic updates. The `equipmentView` component will then send an update request to the server on a regular basis. This attribute is inherited from the JViews Framework faces.
`imageFormat`	The image encoding format, for example "JPG" or "PNG". This attribute is inherited from the JViews Framework faces.
`waitingImage`	Defines the image to be displayed by the equipment faces component while waiting for the image servlet to generate the response image. This attribute is inherited from the JViews Framework faces.
`generateImageMap`	Indicates whether an image map should be generated for the equipment faces component. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapVisible`	Indicates whether the image map should be made visible or not. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapGenerator`	Binds to a bean that subclasses the `ilog.views.servlet.IlvImageMapAreaGenerator` class and is responsible for generating the image map. This attribute is inherited from the JViews Framework faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`imageMapGeneratorClass`	The name of a bean class that subclasses the `ilog.views.servlet.IlvImageMapAreaGenerator` class and is responsible for generating the image map. This attribute is inherited from the JViews Framework

Tag Attributes	Description
	faces. See *Adding an Image Map* in the *Advanced Features of JViews Framework* documentation.
`backgroundColor`	Specifies the background color to be displayed by the equipment faces component when there is no background image. This attribute is inherited from the JViews Framework faces.
`onImageLoaded`	Specifies the JavaScript code to be executed right after a refreshed image is loaded from the server. This attribute is inherited from the JViews Framework faces.
`onCapabilitiesLoaded`	Similar to `onImageLoaded`, but this is executed right after a request for capabilities has been answered by the server. This attribute is inherited from the JViews Framework faces.
`errorMessage`	The error message to be displayed in case of faulty client-server communication. This attribute is inherited from the JViews Framework faces.
`servlet`	Overrides the default image servlet used to generate images. This attribute is inherited from the JViews Framework faces.
`project`	Sets a JViews TGO project to the underlying `IlpEquipment` component.
`dataSource`	Binds the `equipmentView` component with a `dataSource` component. The data source component wraps an `IlpAbstractDataSource` component internally.
`dataSourceId`	Similar to `dataSource`, but binding is done through the data source's unique identifier.
`equipment`	Specifies a custom `IlpEquipment` which replaces the default automatically instantiated equipment component.
`styleSheets`	Specifies a JViews TGO CSS configuration file for the underlying `IlpEquipment` component. It is different from the `styles` tag attribute as it provides server-side configuration, which is specific to JViews TGO. This CSS file may contain component and business data configuration.
`resizable`	Enables or disables the resizing of the equipment faces component. This attribute is inherited from the JViews Framework faces.
`boundingBox`	Defines the width and height of the equipment view component. This attribute is inherited from the JViews Framework faces.
`data`	Defines the data to be displayed, which can be a JViews TGO project, a binding to an `IlpAbstractDataSource` instance, or the unique identifier of a `dataSource` component.
`zoomLevels`	Comma-separated list of fixed zoom levels used by the view. If the zoom levels are not specified, the zoom is bound only by the `maxZoomLevel` property.
maxZoomLevel	The maximum zoom level. This property is used if, and only if, the `zoomLevels` property is not used.

Tag Attributes	Description
	The default value is 10.
tileSize	The size of a tile. If the tile size is greater than or equal to 0, the view will be set in tiled mode. The tile size must be carefully chosen for performance reasons.
tileManager	The tile manager is responsible for retrieving and/or storing image tiles on the server side. The tile manager is used when the view is tiled, that is, if `tileSize` is strictly positive.

Connecting a business data source

To be able to display equipment objects, the equipment faces component must be connected to a data source. This can be done in different ways:

- using a JViews TGO project

  (See *How to set a JViews TGO project to an equipmentView faces component*)

- using the `dataSource` faces component

  (See *How to declare a dataSource faces component for the equipment view* and *How to connect the dataSource faces component to the equipmentView faces component*)

- directly setting an `IlpAbstractDataSource`

  (See *How to set a data source Bean to an equipmentView faces component*)

The easiest way to provide server-side customization and business data to an equipment faces component is through the `project` tag attribute. It allows you to specify a JViews TGO project that will be set to the underlying `IlpEquipment` on the server side. For more information, see Loading a project file . Keep in mind that not all CSS view customizations are supported by the equipment faces component For details, see *Equipment view component services* .

How to set a JViews TGO project to an equipmentView faces component

The following example shows how to pass a JViews TGO project to the `equipmentView` component, and to configure the component dimensions (`width` and `height`) using the `style` tag attribute:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    project="data/myProject.itpr" />
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example2.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The `id` tag attribute defines a unique identifier for the `equipmentView` component. The `context` tag attribute is a binding to a bean defined in the `faces_config.xml` file. The `style` tag attribute defines two CSS properties (`width` and `height`) for the dimensions, in pixels, of the equipment component. The `project` tag attribute is a relative path to a JViews TGO project within the web application. This file should be accessible by the web application.

The following example shows how to declare the `context` bean in the `faces_config.xml` file:

```
<managed-bean>
    <managed-bean-name>contextBean</managed-bean-name>
    <managed-bean-class>
     ilog.tgo.faces.service.IltFacesDefaultContext
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The context should implement the `IlpContext` interface and must use the synchronization strategy `IlSynchronizeOnLockStrategy` in order to address the particular threading issues of a web server.

How to declare a dataSource faces component for the equipment view

Another way to connect business data to the equipment view is through the `dataSource` faces component. This component represents a wrapper for an `IlpAbstractDataSource` object that can be connected to an equipment component. Many different data source components can be declared in a given JSP™ page, but only one can be connected to the equipment view at a time. It is possible to switch data sources dynamically.

The following example shows how to declare a data source in a JSP page:

```
<jvtf:dataSource id="myDataSource" value="#{dataSourceBean}" />
```

The `id` tag attribute defines a unique identifier for the data source component. The `value` tag attribute gets a value binding to a bean previously declared in the `faces_config.xml` file that extends `IlpAbstractDataSource`.

How to connect the dataSource faces component to the equipmentView faces component

Once the data source has been declared, you can connect it to the equipment view as follows:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    dataSourceId="myDataSource" />
```

The `dataSourceId` tag attribute gets the unique identifier of the data source component that will connect it to the equipment view.

The following example shows how to declare the `dataSource` bean in the `faces_config.xml` file:

```
<managed-bean>
  <managed-bean-name>dataSourceBean</managed-bean-name>
  <managed-bean-class>ilog.cpl.datasource.IlpDefaultDataSource</managed-bean
-class>
```

```
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>context</property-name>
    <property-class>ilog.cpl.service.IlpContext</property-class>
    <value>#{contextBean}</value>
  </managed-property>
  <managed-property>
    <property-name>fileName</property-name>
    <property-class>java.lang.String</property-class>
    <value>data/myEquipment.xml</value>
  </managed-property>
</managed-bean>
```

The `dataSource` bean is declared and two properties are set: `context` and `fileName`. The `context` property is set with a value binding to a context bean. It is mandatory, so that the JViews TGO context is consistent across components. The `fileName` property gets a relative path to an XML file compatible with the data source and accessible from the web application.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example3.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to set a data source Bean to an equipmentView faces component

It is also possible to set a data source bean directly to the equipment view component, without requiring the data source component.

For example:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    dataSource="#{dataSourceBean}" />
```

The `dataSource` tag attribute gets a value binding to a bean that extends `IlpAbstractDataSource`. It will connect the equipment component to this data source bean.

How to use the data tag attribute of the equipmentView faces component

The equipment view faces component has a multipurpose `data` tag attribute, which can be used to connect business data sources using:

- a JViews TGO XML project file
- the unique identifier of a data source faces component
- the binding to an instance of `IlpAbstractDataSource`

Note: You must not use any combination of the following tag attributes, which allow you to connect the equipment view to any form of data source:

- `data`
- `dataSourceId`
- `dataSource`
- `project`

When used with JViews TGO projects, the `data` tag attribute behaves exactly like the `project` attribute, getting the relative path to a JViews TGO project, as in the following example:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{myContext}"
                    style="width:740;height:550"
                    data="data/myProject.itpr" />
```

Here `myProject.itpr` is the project file within the web application.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example4.jsp*.

When used with the unique identifier of a data source faces component, the `data` tag attribute behaves exactly like the `dataSourceId` attribute, getting the unique identifier of a data source component, as in the following example:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{myContext}"
                    style="width:740;height:550"
                    data="myDataSource" />
```

Here `myDataSource` uniquely identifies a data source faces component in the current session.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example5.jsp* .

When used with an `IlpAbstractDataSource` instance, the `data` tag attribute behaves exactly like the `dataSource` attribute, getting a value binding to a bean that extends `IlpAbstractDataSource`, as in the following example:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{myContext}"
                    style="width:740;height:550"
                    data="#{dataSourceBean}" />
```

Here `#{dataSourceBean}` is a value binding to the corresponding bean declared in the `faces_config.xml` file.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example6.jsp*.

How to use the binding tag attribute of the equipmentView faces component

Faces components allow you to set a backing bean to replace the default component implementation. So, for the equipment view faces component, the `binding` attribute can be set with a value binding to a backing bean that extends `IltFacesDHTMLEquipmentView` (the DHTML implementation of the equipment view faces component). The following example illustrates this:

```
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    binding="#{myJSFEquipment}" />
```

Here `#{myJSFEquipment}` is a value binding to a backing bean declared in the `faces_config.xml` like this:

```
<managed-bean>
  <description>A bean extending IltFacesDHTMLEquipmentView</description>
  <managed-bean-name>myJSFEquipment</managed-bean-name>
  <managed-bean-class>example.MyEquipmentView</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The backing bean provides more flexibility to the user by giving access to the component API and its instantiation.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example7.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to use the equipment tag attribute of the equipmentView faces component

It is possible to replace the automatically created `IlpEquipment` object with a customized equipment object. This is done with the `equipment` attribute of the equipment view faces component, as follows:

```
<jvtf:equipmentView id="myEquipment"
```

```
                    context="#{contextBean}"
                    width="740"
                    height="550"
                    equipment="#{myIlpEquipment.equipment}" />
```

Here the tag attributes `width` and `height` are used to specify the size of the equipment view. Other examples produce the same results using the `style` tag attribute with the CSS properties "`width`" and "`height`".

In this example, the `equipment` attribute is set with a method that binds to a bean defined in the `faces_config.xml`. The corresponding method (`getEquipment` in this case) will be invoked when the JSP page is parsed. It allows the user to have access to the `IlpEquipment` API as well as to its instantiation. Using the `equipment` attribute and keeping the `IlpEquipment` in a bean is a good way to provide quick access to the underlying `IlpEquipment` API within the web application. Note that the context is not passed to the `myIlpEquipment.getEquipment` method, which means that this bean must be configured with the appropriate context in the `faces_config.xml` file. For example:

```
<managed-bean>
  <description>A bean with read access to the 'equipment' property
  </description>
  <managed-bean-name>myIlpEquipment</managed-bean-name>
  <managed-bean-class>example.MyEquipment</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>context</property-name>
    <property-class>ilog.cpl.service.IlpContext</property-class>
    <value>#{contextBean}</value>
  </managed-property>
</managed-bean>
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example8.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Combining faces components

You can connect components from the core JViews Faces and JViews Framework Faces libraries to the equipment view to combine features and improve user interaction. This is the case with the `overview`, `zoomTool`, `panTool` and `imageButton` components.

How to set up an overview for the equipment view

The `overview` component must be manually set up within the HTML page. Its dimensions and location are important criteria to be considered when designing the HTML page. The following example shows how to declare an overview and connect it to the equipment view:

```
<h:panelGrid columns="2">
  <jvtf:equipmentView id="myEquipment"
                      context="#{contextBean}"
                      style="width:740;height:550"
                      project="data/myProject.itpr" />
  <jvf:overview id="anOverview"
                viewId="myEquipment"
                style="width:123;height:91" />
</h:panelGrid>
```

In the example, an equipment view component is declared with the unique identifier "`myEquipment`" within a two-column `panelGrid`. Then, an overview component is declared so that it is layered after the equipment component. The `viewId` tag attribute is used to connect the equipment view to the overview, through the unique identifier of the main view component. Note that the dimensions of both components are defined in a similar way by the tag attribute `style`.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example9.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to connect a zoom tool and a pan tool to an equipment view

See section The JViews Framework Faces Component Set in the Advanced Features of JViews Framework part of the JViews Diagrammer documentation for details about the zoomTool and panTool.

The following example shows how to attach `zoomTool` and `panTool` components to an equipment view:

```
<h:panelGrid columns="2">
  <jvtf:equipmentView id="myEquipment"
                      context="#{contextBean}"
                      style="width:740;height:550"
```

```
                      project="data/myProject.itpr" />
  <h:panelGrid columns="1">
    <jvf:panTool id="aPanTool"
                 viewId="anEquipment"
                 style="width:123;height:123" />
    <jvf:zoomTool id="aZoomTool"
                  viewId="anEquipment"
                  style="width:123;height:322" />
  </h:panelGrid>
</h:panelGrid>
```

In this example, an equipment view component is declared with the unique identifier "`myEquipment`" within a two-column `panelGrid`. Then, a new one-column `panelGrid` is declared to accommodate the `panTool` and `zoomTool` components. The `viewId` tag attribute is used to connect the equipment view to the other components. Note that the `style` tag attribute is used to set the dimensions for all the declared components.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example10.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to add image buttons and set client-side actions for the equipment view component

Although `zoomTool` and `panTool` components provide basic user interaction, you can also set client actions to image buttons to achieve similar results. The advantage is that image buttons are more customizable, as the user can define the action to be set. The following example shows how to declare image buttons and associate them with client-side actions.

```
<!-- Create a 2 columns grid -->
<h:panelGrid columns="2">

  <!-- Declare a button for zooming in -->
  <jv:imageButton onclick="myEquipment.zoomIn(true)"
                  image="images/zoom.gif"
                  rolloverImage="images/zoomh.gif"
                  selectedImage="images/zoomd.gif"
                  title="Zoom In"
                  message="Zoom In" />

  <!-- Declare a button for zooming out -->
  <jv:imageButton onclick="myEquipment.zoomOut(true)"
                  image="images/unzoom.gif"
                  rolloverImage="images/unzoomh.gif"
                  selectedImage="images/unzoomd.gif"
                  title="Zoom Out"
                  message="Zoom Out" />
</h:panelGrid>
<jvtf:equipmentView id="myEquipment"
```

```
                        context="#{contextBean}"
                        style="width:740;height:550"
                        project="data/myProject.itpr" />
```

This example declares two image buttons:

- one for zooming in
- one for zooming out

Each button declaration defines the following attributes:

- `onclick`: The JavaScript™ action to be triggered when the button is pressed.
- `image`: The main button image.
- `rolloverImage`: The image to be displayed when the mouse pointer rolls over the button.
- `selectedImage`: The image to be displayed when the button is pressed.
- `title`: The tooltip message displayed when the mouse pointer stays over the button.
- `message`: The message displayed in the `messageBox` component when the mouse pointer stays over the button.

The `onclick` tag attribute is the most important as it defines the action associated with the button. Note that it uses the JavaScript API of the `equipmentView` component to perform the desired action:

- `onclick="myEquipment.zoomIn(true)"`: This uses the `zoomIn` JavaScript call to zoom in the equipment view component.
- `onclick="myEquipment.zoomOut(true)"`: This uses the `zoomOut` JavaScript call to zoom out the equipment view component.

The `onclick` attribute can be set with any valid JavaScript code, which will be executed when the button is pressed. The other tag attributes define the look and feel of the button, with corresponding images and tooltip text.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example11.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Interacting with the equipment view component

JViews Framework Faces and core JViews Faces libraries declare predefined interactors that can be connected to the `equipmentView` component to add extra user interaction. Interactors are faces components that execute client- or server-side actions. Most of them can be extended and configured to suit the user needs.

How to declare an interactor and connect it to the equipment view component

The following example shows how to declare a predefined interactor (the `pan` interactor) in the JSP™ page and connect it to the `equipmentView` component so that it is always available.

```
<!-- Declare the predefined 'pan' interactor -->
<jvf:panInteractor id="pan" />

<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    interactorId="pan"
                    project="data/myProject.itpr" />
```

In this example, the predefined `panInteractor` is declared. A unique identifier is associated with it ("`pan`"). Then, the `interactorId` tag attribute of the `equipmentView` component specifies the interactor to be connected to the equipment view.

How to associate interactors with image buttons in the equipment view component

Usually many interactors are made available in a web application. The following example shows how to declare multiple predefined interactors and how to use image buttons to make them active. Note that only one interactor can be set in the equipment view component at a time. Whenever a new interactor is set, the previous one is removed.

```
<!-- Declare the predefined 'select' interactor -->
<jvtf:selectInteractor id="select" />

<!-- Declare the predefined 'pan' interactor -->
<jvf:panInteractor id="pan" />

<!-- Create a 4 columns grid -->
<h:panelGrid columns="4">

  <!-- Declare a button for selection -->
  <jv:imageButton onclick="myEquipment.setInteractor(select)"
                  buttonGroupId="interactors"
                  image="images/arrow.gif"
                  rolloverImage="images/arrowh.gif"
```

```
                    selectedImage="images/arrowd.gif"
                    title="Select Interactor"
                    message="Select Interactor" />

  <!-- Declare a button for panning -->
  <jv:imageButton onclick="myEquipment.setInteractor(pan)"
                    buttonGroupId="interactors"
                    selected="true"
                    image="images/pan.gif"
                    rolloverImage="images/panh.gif"
                    selectedImage="images/pand.gif"
                    title="Pan Interactor"
                    message="Pan Interactor" />

  <!-- Declare a button for zooming in -->
  <jv:imageButton onclick="myEquipment.zoomIn(true)"
                    image="images/zoom.gif"
                    rolloverImage="images/zoomh.gif"
                    selectedImage="images/zoomd.gif"
                    title="Zoom In"
                    message="Zoom In" />

  <!-- Declare a button for zooming out -->
  <jv:imageButton onclick="myEquipment.zoomOut(true)"
                    image="images/unzoom.gif"
                    rolloverImage="images/unzoomh.gif"
                    selectedImage="images/unzoomd.gif"
                    title="Zoom Out"
                    message="Zoom Out" />
</h:panelGrid>
<jvtf:equipmentView id="myEquipment"
                      context="#{contextBean}"
                      style="width:740;height:550"
                      interactorId="pan"
                      project="data/myProject.itpr" />
```

This example defines two predefined interactors:

- `selectInteractor`: This is a server-side interactor that processes object selection by default. (See *The selectInteractor faces component* for details.)
- `panInteractor`: This is a client-side interactor that enables panning of the image displayed by the equipment view component.

Two buttons are declared to connect the interactor to the equipment view component. The `buttonGroupId` tag attribute is used to group image buttons so that only one button of the group is selected at a time. The `selected` attribute is used to specify which button should be made selected when the page is loaded. This should correspond to the interactor initially connected to the equipment view with the `interactorId` tag attribute. In this case, the pan button is selected (`select="true"`) and the `pan` interactor is connected to the equipment view (`interactorId="pan"`).

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example13.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The selectInteractor faces component

The `selectInteractor` faces component has been defined as an interactor that maps client-side mouse clicks to server-side events dispatched to the underlying view interactor. It extends the JavaServer™ Faces `UICommand` component, which means that it will fire `ActionEvents` to registered `ActionListeners`.

This component allows you to create customized `IlvManagerViewInteractor` instances that will process the mouse actions on the client side. By default, it uses the `IltSelectInteractor`, which allows selecting, dragging and expanding graphic objects.

This interactor has the following limitations in terms of handling events:

- Only `BUTTON1` and `BUTTON3` mouse buttons are supported (left and middle).
- Pop-up menus are supported in a different way from the Swing equipment component. See *Adding pop-up menus*.
- Double-click events are not supported.
- There is no visual feedback when dragging an object (the graphic representation of the object does not follow the mouse pointer).
- No keyboard actions are supported except the following modifiers:
  - `Shift` key
  - `Control` key
  - `Alt` key
  - `Meta` key

Client-side interactions are converted into the following mouse events, dispatched to the appropriate interactor:

- `MOUSE_PRESSED`
- `MOUSE_DRAGGED`
- `MOUSE_RELEASED`

How to declare the selectinteractor faces component for the equipment view

The `selectInteractor` is declared like any other faces component defined in the JViews TGO faces tag library:

```
<jvtf:selectInteractor id="select" />
```

Configuring the selectInteractor

The `selectInteractor` faces component is configured through the following tag attributes:

Tag Attributes of the `selectInteractor` Faces Component

Tag Attributes	Description
`cursor`	Defines the mouse cursor to be used when the interactor is active; it should be one of the cursors supported by the web browser.
`lineWidth`	Defines the width of the interaction area drawn by the interactor.
`lineColor`	Defines the color of the interaction area drawn by the interactor.
`actionName`	Defines the name of the action event triggered by this interactor; it is used to identify events coming from this interactor.
`autoSubmit`	Defines whether the request will be submitted by a mouse click or not; this tag is used to control when the actions are submitted. When this tag attribute is set to `false`, the `selectInteractor` will not submit any request after the user interaction, but wait until some other component does it.
`actionListener`	Defines an action listener that is called when this interactor is used. Action listeners should implement the `ActionListener` interface (from JSF library), but JViews TGO faces provide the `IltFacesGraphInteractorActionListener` abstract implementation that decodes the user interactions into events dispatched to a given view interactor. Subclasses should implement the method `getViewInteractor` in order to return the appropriate view interactor to process the events. The default implementation (`IltFacesSelectInteractorListener`) dispatches all events to the `IltSelectInteractor` view interactor. Therefore, the `actionListener` tag attribute may be used to register any `ActionListener` that will be notified whenever a user interaction has been performed, or a subclass of `IltFacesGraphInteractorActionListener` can be registered to decode the user interaction into events dispatched to view interactions (`IlpViewInteractor`).
`invocationContext`	Defines whether the server-side processing is performed in the JSF lifecycle or directly by the image servlet. The possible values are: ♦ JSF_CONTEXT: Processing is done in the JSF lifecycle (the default value) ♦ IMAGE_SERVLET_CONTEXT: Processing is done by the image servlet, bypassing the JSF lifecycle The `selectInteractor` submits requests to be processed on the server side. By default, the request is addressed to the JavaServer™ Faces controller servlet which processes all requests according to the well-defined JSF lifecycle. This means that all component dependencies will be verified, any registered listener will be notified. The result is a full page refresh with an update of all components involved. If your interaction triggers updates of components other than the `equipmentView`, then the `JSF_CONTEXT` should be used.

Tag Attributes	Description
	If, on the other hand, your requests are supposed to only update the image displayed by the `equipmentView` component, then you may want to benefit from the faster `IMAGE_SERVLET_CONTEXT`. In this case, the interactor requests will be addressed to the image servlet responsible for generating the image displayed by the `equipmentView` component. Note that the image servlet has no access to any faces component other than `equipmentView`, which means that your request cannot rely on other faces components, and that any registered listener for changes on the `selectInteractor` and `equipmentView` will not be updated. There is one exception to this rule. As the `overview` faces component is largely used with the `equipmentView` faces component, and as its displayed image is also generated by the image servlet, you can force the overview to be refreshed whenever the main view (`equipmentView`) is updated, at the cost of an extra client-server-client roundtrip. To do so, you must set the `autoRefresh` tag attribute of the `overview` faces component to true.
`binding`	Defines a binding expression to a backing bean.

The `cursor`, `lineWidth` and `lineColor` tag attributes control the look of the interactor when it is activated, they do not affect its functionality.

If you have started the bundled Tomcat web server, the following links will take you to the small samples illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example12.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html***.

where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring action listeners for the select interactor component

Action listeners are responsible for processing the interactions performed with the select interactor component. Their default behavior is to convert the interactions into server events dispatched to the `IltSelectInteractor`. It is possible to override this behavior by adding action listeners to the component.

Unlike in the regular equipment Swing component, it is not possible to declare view interactors through a CSS file. Instead, in the equipment faces component, the view interactors are declared within customized action listeners added to the `selectInteractor` faces component.

As the `selectInteractor` extends the JavaServer™ Faces `UICommand`, it allows one or more action listeners (implementing `javax.faces.event.ActionListener`) to be registered to receive events (`javax.faces.event.ActionEvent`) whenever a user interaction is performed. For the `ActionEvent` API there are the following methods:

- `getComponent()` or `getSource()`: Return a reference to the interactor faces component that is currently active (for example, `IltFacesGraphInteractor`).

A predefined abstract implementation of the `ActionListener` interface named `IltFacesGraphInteractorActionListener` is provided to translate client-side interactions into server-side events that are dispatched to a given view interactor. When notified, this

class translates user interactions into mouse events that are automatically dispatched to the `IlpViewInteractor` returned by the abstract method `getViewInteractor(actionName)`.

The following example illustrates how to override the default `selectInteractor` behavior with a customized one:

```
<jvtf:selectInteractor id="select"
                       actionListener="#{MyListenerBean}"
                       invocationContext="JSF_CONTEXT" />
```

Here the `actionListener` tag attribute gets a binding to a bean implementing the `javax.faces.event.ActionListener` interface. Note that `actionListener` will override the default behavior of the `selectInteractor`. It is possible to add more than one action listener, combining customized action listeners with the default behavior as shown in the next example:

```
<jvtf:selectInteractor id="select"
                       invocationContext="JSF_CONTEXT">
  <f:actionListener
   type="ilog.tgo.faces.graph.dhtml.event.IltFacesSelectInteractorListener"/
>
  <f:actionListener type="demo.MyInteractionListener"/>
</jvtf:selectInteractor>
```

Here `IltFacesSelectInteractorListener` is the action listener (extends `IltFacesGraphInteractorActionListener`) that implements the default behavior of the `selectInteractor` faces component, and `MyInteractionListener` is a customized implementation of the `javax.faces.event.ActionListener` interface. The `actionListener` tag is used to add several action listeners to the `selectInteractor`, which are invoked in the order in which they have been declared. Note that action listeners may conflict with each other, especially multiple implementations of `IltFacesGraphInteractorActionListener`, as the first one invoked may change the business model and invalidate the next action listener.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to customize action listeners: *http://localhost:8080/jsf-equipment-step-by-step/faces/example14.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

The clientSelectInteractor faces component

The `clientSelectInteractor` faces component is an interactor designed to minimize the number of image requests and image updates between the graph view on the client and the image servlet on the server by dynamically rendering and managing the selection borders in the client side.

Instead of requesting a new graph view image every time the user selects an object, the `clientSelectInteractor` dynamically renders an HTML rectangular selection around the object. The server is notified so that the selection model is kept synchronized with the user interactions. A new graph view image is requested only when the user drags objects or interacts with specific decorations (such as information icons, and expansion icons).

The performance and responsiveness is greatly improved as lesser images are generated and dispatched by the server. However, the selection graphic feedback is impacted, as the client is limited to only displaying a rectangular border around the selected object.

The interactor can be configured to work in *image* mode. In this mode, it will ask the server to process the selection and get a new image on every user interaction. The dynamic selection border will only be displayed while objects are being dragged, as when objects are in their resting position the new image sent by the server already represents the selection.

It is possible to customize the interaction with object decorations, controlling when a click on a particular object decoration should trigger a new image request or not.

Note: Unlike the `selectInteractor`, the `clientSelectInteractor` always communicates with the image servlet directly. Therefore, it does not follow the JSF lifecycle, which means that only the view faces component and a possibly attached overview are updated and not all the other components in the page.

How to declare the clientSelectInteractor faces component for the equipment view

The `clientSelectInteractor` is declared like any other faces component defined in the JViews TGO faces library:

```
<jvtf:clientSelectInteractor is="clientSelect" />
```

Configuring the clientSelectInteractor

The `clientSelectInteractor` faces component is configured through the following tag attributes:

Tag Attributes of the `clientSelectInteractor` Faces Component

Tag Attributes	Description
`binding`	Defines a binding expression to a backing bean, allowing the user to customize the `clientSelectInteractor` faces component.
`message`	Defines the message displayed by the view when the interactor is set.
`cursor`	Defines the mouse cursor to be used when the interactor is active. It should be one of the cursors supported by the web browser.
`menuModeId`	The identifier passed to dynamically generated popup menus.
`moveAllowed`	Specifies whether this interactor allows mouse dragging.
`objectActionMethodBinding`	Defines a method binding to process actions on the object or on specific decorations attached to it.
`onSelectionChanged`	Specifies a JavaScript handler to be called whenever the selection changes. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`imageMode`	Sets the interactor to image mode. A new image will be requested every time the user interacts with the view. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`lineWidth`	The width of the selection border dynamically rendered by the interactor. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`lineColor`	The color of the selection border dynamically rendered by the interactor. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`forceUpdateProperties`	Forces the request of additional information when the interactor is in image mode. Used in conjunction with the `infoProviderMethodBinding`. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.
`infoProviderMethodBinding`	Defines the information provider method binding to return additional information about the selected object. Deprecated in JViews TGO 8.0. See *The selectionManager faces component* for an alternative.

If you have started the bundled Tomcat web server, the following link will take you to a small sample illustrating this: *http://localhost:8080/jsf-equipment-step-by-step/faces/example19.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Configuring an object action for the clientSelectInteractor

By default, the `clientSelectInteractor` supports interactions with the following object decorations:

- The information icon

- The system icon
- The expand and collapse icons

If the user clicks on any of these decorations, the interactor triggers a default object action instead of selecting the object. These actions change the look of the objects, which means that a new image is generated.

It is possible to override or even extend this behavior through the `objectActionMethodBinding` tag attribute, as follows:

```
<jvtf:clientSelectInteractor id="clientSelect"

objectActionMethodBinding="#{interactorBean.objectAction}" />
```

Here, the `objectActionMethodBinding` tag attribute is bound to the `objectAction` method declared in the `interactorBean`. The method binding must conform to the following signature:

```
boolean methodName(IlpGraphView, int, int)
```

The `IlpGraphView` is a reference to the graph view containing all the objects. The two integer parameters are the `x` and `y` components of the view position where the user has clicked. Returning `true` indicates that a new image has to be generated, while `false` indicates that nothing has been processed and the clicked object will be selected.

In the example above, the `objectAction` method will override the default object action behavior (which is to react on expand, collapse, information and system icons). It is possible to extend this default behavior by overriding the method `processObjectAction` in class `IltFacesDefaultObjectAction`.

The selectionManager faces component

You can customize the way the selection is performed and displayed by using a selection manager.

This selection manager is defined in a facet on the `equipmentView` tag, as follows:

```
<jvtf:equipmentView id="tgoViewId" interactorId="select">
  <f:facet name="selectionManager">
      <jvtf:selectionManager imageMode="false" [...] />
  </f:facet>
</jvtf: equipmentView >
```

The selection manager has two display modes:

- image (default)

  The image is refreshed after each selection. A new image is requested to the server at each selection which allows the client to get nice selection graphics.

- regular

Rectangles representing the selection are displayed on top of the view. The roundtrip to the server is minimal: the generation of a new image is not required and the response time is faster but the selection feedback is limited to a selection rectangle.

Tip: Image Mode versus Regular Mode

Using one mode rather than the other depends on your criteria: performance or graphic feedback. Image mode provides a better graphic feedback but is slower because of the image generation and the need for an extra request to get additional information about the selection on the client. Regular mode offers basic graphic feedback but better performance.

Other parameters can be configured on the `selectionManager`, like for example the line width or the color of the selection rectangle used in regular selection mode:

```
<jvtf:selectionManager lineWidth="2" lineColor="red"/>
```

Note: The selection manager currently supports integration with the following TGO Faces interactor: `clientSelectInteractor`.

Configuring the selectionManager

The `selectionManager` faces component is configured through the following tag attributes:

Tag Attributes of the `selectionManager` Faces Component

Tag Attributes	Description
`infoProviderMethodBinding`	A method binding that respects the signature List methodName(IlpGraphView, IlpRepresentationObject). The returned value of this method is a list of additional properties associated with the selected object. A valid item of this list is a String or a list itself. As of JViews TGO 8.0, the preferred way to transfer object properties to client is via the `propertyAccessor` tag of the `selectionManager`.
`binding`	The value binding expression linking this component to a property in a backing bean. If this attribute is set, the tag does not create the

Tag Attributes	Description
	component itself but retrieves it from the Bean property. This attribute must be a value binding.
`lineColor`	The color of selection rectangles lines.
`forceUpdateProperties`	Force to make additional request to query the current selection and additional properties in image mode to enable client-side selection listener.
`id`	The ID of this component.
`imageMode`	The image mode. In image mode, the image is refreshed on each selection. In regular mode, only the selected object(s) bounding box is queried and rectangles are dynamically displayed on top of the view. Note that the client-side listeners on selection and additional information on selected objects are available in image mode if and only if the `forceUpdateProperties` property is set to true. In regular mode no special configuration is needed. By default the manager is in image mode.
`lineWidth`	The width of selection rectangle lines.
`onSelectionChanged`	A JavaScript handler called when the selection has changed. The handler can use the predefined variable '`selection`' which is the list of current selected items. To use this handler the `selectionManager` must be in regular mode or the `forceUpdateProperties` must be set if in image mode. Refer to the user's documentation for further information.
`propertyAccessor`	The reference to the value binding expression to an `IltFacesPropertyAccessor` instance that will be used to access model properties of the selected objects.
`fillOn`	true to display filled selection rectangles. The fill color is the line color with a transparency of 50%.

Exposing selection details

You can expose details on the current selection by taking advantage of the property accessor of the selection manager.

The `IltFacesPropertyAccessor` contains several methods that can be overridden to configure or specialize the way it gives access to model properties. In particular, you can filter the properties that are exposed to clients by overriding the following method:

```
List getPropertyNames(IlpGraphicView view, IlpRepresentationObject object)
```

The following code illustrates how to provide your property accessor:

```
<jvtf:selectionManager propertyAccessor="{#serverBean.propertyAccessor}" [...
]
/>
```

The following code illustrates how to implement your custom requirements in a new property accessor:

```
public class ServerBean {

  private IltFacesPropertyAccessor accessor = new MyPropertyAccessor();

  public IltFacesPropertyAccessor getPropertyAccessor() {
    return accessor;
  }

  class MyPropertyAccessor extends IltFacesPropertyAccessor {

   protected List getPropertyNames(IlpGraphicView view, IlpRepresentationObject
 object) {
      […]
    }
  }
}
```

Then you can register a JavaScript listener that will be called when the selection changes:

```
<jvtf:selectionManager onSelectionChanged="displayProperties(selection)"/>
```

The JavaScript function can be as follows:

```
// Alert the ID and bounds of all the selected objects
function displayProperties(selection) {
  for (var i = 0; i < selection.length; i++)
   alert(selection[i].getID()+"+selection[i].getBounds());
}
```

In addition to the ID and bounds properties of the selected object, you can also expose the properties of the selected object in the TGO model as follows:

```
// Alert all the properties of all the selected objects
function displayProperties(selection) {
  for (var i = 0; i < selection.length; i++) {
    var propertiesNames = selection[i].getObjectPropertyNames();
    for (var j = 0; j < propertiesNames.length; j++)
         alert(selection[i].getObjectProperty(propertiesNames[j]));
  }
}
```

Note: To obtain selected object properties information on the client side while you are running the selection in image mode, you need to force an additional request by setting the property `forceUpdateProperties` to `true`. In regular mode this feature is available without any overhead.

If you have started the bundled Tomcat Web server, the following link will take you to the small sample illustrating how to use the selection manager with property accessors: *http://localhost:8080/jsf-equipment-step-by-step/faces/example20.jsp*.

You can find more information about the sample Web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Managing object selection

The `selectionManager` also allows for changing the selection state of objects programmatically on the client side by means of JavaScript.

This can be accomplished by using the following API:

- `selectById(id, extend)`
- `selectAll()`
- `deselectAll()`

Note: In all cases, the object must be selectable in order to get selected.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to use the selection manager API: *http://localhost:8080/jsf-equipment-step-by-step/faces/example22.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Selecting and deselecting an object

The `selectById` function allows you to select or deselect an object by providing the object's identifier:

```
equipmentView.getSelectionManager().selectById("tgoObjectId");
```

This method call will select the object with the identifier `tgoObjectId` and deselect the object currently selected.

You can extend/reduce the current selection by selecting/deselecting a node as follows:

```
equipmentView.getSelectionManager().selectById("tgoObjectId", true);
```

This method call keeps the existing selection and selects the object with the identifier `tgoObjectId` if it is not already selected, or deselects it if it is already selected.

Selecting all objects

The `selectAll` function allows you to select all objects:

```
equipmentView.getSelectionManager().selectAll();
```

This method call selects all visible objects.

Deselecting all objects

The `deselectAll` function allows you to deselect all objects:

```
equipmentView.getSelectionManager().deselectAll();
```

This method call clears the selection of all visible objects.

Zoom constraints

When the zoom level is equal to 1, the manager content is adjusted to the bounds of the JSF view so as to be displayed entirely. Consequently, a zoom level of `n` means that the content is scaled by a factor of `n`. For example, a zoom factor of 2 means that the manager content is displayed double its size.

By default, the view is constrained by the manager content bounds. The direct consequences are that:

- Pan actions or zoom interactions cannot go out of the manager content bounds.
- The view zoom level cannot be lower than 1.

This constraint can be removed by setting the `constrainedOnContents` property to `false`, as follows:

```
<jvtf:equipmentView constrainedOnContents="false" [...] />
```

The zoom level applied to the view by using the zoom interactor of JavaScript™ zoom actions can be free or constrained to specified zoom levels. In the free zoom mode, the only constraints are the minimum and maximum zoom levels. The default value of the minimum zoom level is set to 1 and the default value of the maximum zoom level is set to 10. These constraints can be customized with the `minZoomLevel` and the `maxZoomLevel` properties respectively.

```
<jvtf:equipmentView minZoomLevel="2" maxZoomLevel="20" [...] />
```

Note: By default, the minimum zoom level cannot be lower than 1.

To specify fixed zoom levels, use the `zoomLevels` property, as follows:

```
<jvtf:equipmentView zoomLevels="1.0, 2.0, 5.0, 10.0" [...] />
```

When this property is set:

- The `minZoomLevel` and `maxZoomLevel` properties are ignored.
- The `minZoomLevel` becomes the first zoom level and the `maxZoomLevel` the last zoom level in the list.
- The zoom interactor will fit to the nearest zoom level.
- The built-in zoom actions on the JavaScript view proxy use these fixed zoom levels.

Fixed zoom levels must be used in order for a tiled view to be cached on the client-side.

For more details on setting up zooming in the equipment view, see *How to associate interactors with image buttons in the equipment view component*.

Controlling the displayed area

The Equipment Faces component allows developers to specify the area that will be displayed on the client. For example, this enables developers to set the initial visible area or possibly to change at runtime the clipping rectangle so that it centers or focuses on a given equipment element.

This can be done by means of the `boundingBox` property as follows:

```
<jvtf:equipmentView [...] boundingBox="0,0,100,200"/>
```

The value provided corresponds to the `x`, `y`, `height` and `width` of the area of interest in manager coordinates separated by commas.

Programmatically, this property can be used during a JSF action to reset or modify the visible area by providing an instance of `IlvRect` as illustrated below.

```
public class ActionProvider {
  [...]

  public void changeAreaDisplayed() {
    IltFacesEquipmentView facesEquipmentView = …;
    facesEquipmentView.setBoundingBox(new IlvRect(0,0,100,100));
  }
}
```

Adding pop-up menus

Unlike the equipment Swing component, the equipment faces component does not rely on the `IlpPopupMenuFactory` interface to declare contextual menus. Instead, it is based on the `contextualMenu` tag defined in the `jviews-framework-faces.tld` tag library descriptor. This means that pop-up menus in equipment faces cannot be declared in CSS files.

The `contextualMenu` tag allows you to define two distinct types of pop-up menu:

- Static pop-up menus: The menu structure is hard coded in the JSP™ file, it applies to all objects and cannot be changed dynamically.
- Dynamic pop-up menus: The menu structure is defined by the `IlvMenuFactory` interface and can be created dynamically where the pop-up was activated

Note: In JViews TGO Faces, the pop-up menu does not trigger any object selection, that is, the object right below the mouse pointer is not automatically included in the selection model.

How to add a static pop-up menu to an equipment faces component

The static pop-up menu is fully declared within the JSP file, using the following tags:

- `contextualMenu` (`jviews-framework-faces.tld` library)
- `menu` (`jviews-faces.tld` library)
- `menuItem` (`jviews-faces.tld` library)
- `menuSeparator` (`jviews-framework-faces.tld` library)

The following example illustrates how to declare a static pop-up menu within an equipment faces component:

```
<!-- Declare the Equipment Faces component -->
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    style="width:740;height:550"
                    project="data/default_project.xml">
  <!-- Declare the contextual menu -->
  <jvf:contextualMenu>
    <!-- Declare the root popup menu -->
    <jv:menu label="root">
      <jv:menuItem label="Zoom In"
                   image="images/zoom.png"
                   onclick="myEquipment.zoomIn()" />
      <jv:menuItem label="Zoom Out"
                   image="images/unzoom.png"
                   onclick="myEquipment.zoomOut()" />
```

```
        <jv:menuSeparator />
        <jv:menuItem label="Fit To Contents"
                     image="images/zoomfit.png"
                     onclick="myEquipment.showAll()" />
        <jv:menuItem label="Alert!"
                     image="images/alert.png"
                     onclick="alert('Alert menu item!')" />
      </jv:menu>
    </jvf:contextualMenu>
  </jvtf:equipmentView>
```

In this example, the `contextualMenu` tag is declared within the equipment faces component declaration (`equipmentView` tag). It is structured as a root menu (`menu` tag) with multiple menu items (`menuItem` tags).

The `onclick` attribute in the `menuItem` tag is the most important. It defines the JavaScript™ code to be executed when the menu item is selected. See `index` for details on the available tag attributes.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to declare a static pop-up menu: *http://localhost:8080/jsf-equipment-step-by-step/faces/example15.jsp* .

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to add and customize a dynamic pop-up menu for an equipment faces component

Like the static pop-up menu, the dynamic pop-up menu is declared in a JSP page using the `contextualMenu` tag inside the equipment faces declaration (`equipmentView` tag). However, instead of declaring the menu structure, it declares a menu factory (implementing the `IlvMenuFactory` interface) that is invoked whenever the pop-up menu is activated. The following exampe illustrates how the dynamic pop-up menu is declared:

```
<head>
  <!-- Specify a CSS file -->
  <link href="data/style.css" rel="stylesheet" type="text/css"/>
</head>

<!-- Declares a select interactor, which will be attached to the view -->

<jvtf:selectInteractor id="select"
                       menuModelId="selectInteractor"
                       invocationContext="IMAGE_SERVLET_CONTEXT" />

<!-- Declare the Equipment Faces component -->
<jvtf:equipmentView id="myEquipment"
                    context="#{contextBean}"
                    interactorId="select"
                    backgroundColor="#F5F5F5"
```

```
                        style="width:740;height:550"
                        project="data/default_project.xml">
      <!-- Declare the contextual menu with given popup menu factory -->
      <jvf:contextualMenu factory="#{popupMenuFactory}"
                          itemStyleClass="menuItem"
                          itemHighlightedStyleClass="menuItemHighlighted"
                          itemDisabledStyleClass="menuItemDisabled" />
    </jvtf:equipmentView>
```

As shown above, the `contextualMenu` tag is used within the `equipmentView` declaration to add a pop-up menu to the equipment faces component. In addition, the following tag attributes are noteworthy:

The factory tag attribute

This attribute of the `contextualMenu` tag is bound to a bean implementing the `IlvMenuFactory` interface, which defines one single method:

```
  public IlvMenu createMenu(Object graphicComponent, Object selectedObject,
String menuModelId);
```

When this method is automatically called, the `graphicComponent` attribute refers to the underlying graphic view (`IlpGraphView`, superclass of `IlpEquipmentView`). It allows full access to the `IlpEquipmentView` API, including selection model, controller, and so on.

The `selectedObject` attribute refers to the representation object (`IlpRepresentationObject`) located immediately below the mouse pointer when the pop-up menu was activated, if any. Note that this object may or may not be selected. It is independent of the selection model.

The `menuModelId` corresponds to the value set in the `menuModelId` tag attribute of the `selectInteractor` tag. It allows you to create custom pop-up menus based on the active interactor.

The following `IlvMenuFactory` example creates a basic pop-up menu:

```
  public IlvMenu createMenu(Object graphicComponent,
                            Object selectedObject,
                            String menuModelId) {
    // Create the root menu
    IlvMenu root = new IlvMenu("Root");

    // Create 3 JavaScript actions
    ActionListener jsAction = new
JavaScriptActionListener("myEquipment.zoomIn()");
    root.addChild(new IlvMenuItem("Zoom in", jsAction,
                                  "images/zoom.png", true));

    jsAction = new JavaScriptActionListener("myEquipment.zoomOut());
    root.addChild(new IlvMenuItem("Zoom out", jsAction,
                                  "images/unzoom.png", true));

    jsAction = new JavaScriptActionListener("alert('Alert menu item!')");
```

```
    root.addChild(new IlvMenuItem("Alert!", jsAction,
                                  "images/alert.png", true));

    return root;
  }
```

In this example, `IlvMenu` is the root menu that contains menu items (`IlvMenuItem`). Each menu item has an `ActionListener` associated with it. In this case, the predefined `JavaScriptActionListener` class is used to trigger JavaScript code executed on the client when the corresponding menu item is activated. Note that `myEquipment` in `myEquipment.zoomOut()` refers to the identifier of the `equipmentView` faces component from the previous example. `zoomOut()` is the JavaScript method that performs zooming out on the client side.

The itemStyleClass, itemHighlightedStyleClass and itemDisabledStyleClass tag attributes

These attributes of the `contextualMenu` tag are used to customize the look of the pop-up menu. They declare the CSS classes that contain styling definitions for items, highlighted items and disabled items, respectively as follows (from the `style.css` file):

```
.menuItem {
  background: #E5E5E5;
  font-family: sans-serif;
  font-size: 14px;
  font-style: normal;
  color: black;
}

.menuItemHighlighted {
  background: #FFE5A5;
  font-style: normal;
  color: black;
}

.menuItemDisabled {
  font-style: italic;
  color: #A5A5A5;
}
```

The menuModelId tag attribute

This attribute of the `selectInteractor` tag is used by the menu factory to identify which pop-up menu to create based on the interactor that is currently active.

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to customize pop-up menus: *http://localhost:8080/jsf-equipment-step-by-step/faces/example16.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

How to trigger server actions from a dynamic pop-up menu of an equipment faces component

The dynamic pop-up menu can trigger two types of action:

- client actions: JavaScript actions executed on the client
- server actions: Java™ actions executed on the server

When building the dynamic menu, the pop-up menu factory (`IlvMenuFactory`) creates a root menu (`IlvMenu`) with menu items (`IlvMenuItem`) and each menu item has an action listener (`ActionListener`) associated with it.

Client actions are defined by the predefined `JavaScriptActionListener`, which has been described in the previous example.

Server actions, like interactions, can be processed either by the JavaServer™ Faces lifecycle or directly by the image servlet. This is defined by an invocation context that can be either one of the following:

- `JSF_CONTEXT`: Processing takes place in the JSF lifecycle (default value)
- `IMAGE_SERVLET_CONTEXT`: Processing is performed by the image servlet, bypassing the JSF lifecycle.

Server actions are defined by subclassing the `FacesViewActionListener` abstract class. The subclass should define the desired invocation context and implement the `public void actionPerformed(EventObject event)` method. The event parameter is in fact an instance of the `ServletActionListener` class that has the following convenient methods in its API:

- `getGraphicComponent()`: This method returns the underlying view (instance of `IlpEquipmentView`)
- `getObject()`: This method returns the representation object (`IlpRepresentationObject`) located right below the mouse pointer when the pop-up menu was activated)

This allows full access to the `IlpEquipmentView` API, including selection model, controller, and so on.

The following example shows a basic subclass of `FacesViewActionListener`:

```
public class MyActionListener extends FacesViewActionListener {
  /**
   * Constructor. Sets the invocation context.
   */
  public AddAlarmActionListener() {
    super(IlvDHTMLConstants.IMAGE_SERVLET_CONTEXT);
  }

  /**
   * Access the equipment view and the active object.
   *
   * @param event An instance of ServletActionListener.
   */
  public void actionPerformed(EventObject event) throws Exception {
    ServletActionEvent saEvt = (ServletActionEvent)event;
```

```
    // access the equipment view
    IlpEquipmentView view = (IlpEquipmentView)saEvt.getGraphicComponent();

    // access the active object
    IlpObject obj = (IlpObject)saEvt.getObject();

    // implement your action with 'view' and 'obj'
  }
}
```

Once the action listener has been defined, it can be used within the pop-up menu factory (`IlvMenuFactory`) as follows:

```
public IlvMenu createMenu(Object graphicComponent, Object selectedObject,
                          String menuModelId) {
  // Create the root menu
  IlvMenu root = new IlvMenu("Root");

  // Create one server action
  ActionListener srvAction = new MyActionListener();
  root.addChild(new IlvMenuItem("My action", srvAction,
                                "images/action.png", true));

  return root;
}
```

If you have started the bundled Tomcat web server, the following link will take you to the small sample illustrating how to handle server actions: *http://localhost:8080/jsf-equipment-step-by-step/faces/example17.jsp*.

You will find more information about the sample web application in: ***<installdir>/samples/faces/jsf-equipment-step-by-step/index.html*** where `<installdir>` stands for the directory where IBM® ILOG® JViews TGO is installed.

Tiling

The Equipment Faces component provides support for tiling. The tiling support consists of providing developers with the ability of configuring the Equipment Faces component to compute, cache and provide on demand only the areas of its graphical representation that are visible to the client at one given time, instead of computing and providing the entire area of the graphical representation that may not be visible to the client.

Note: The tiling support provided by the JViews TGO Equipment Faces component is based on the tiling support by the underlying JViews Faces Framework. Any difference in the default behavior as defined in the JViews Faces Framework documentation is documented in this section.

Configuration

Tiled view

See *Concepts* in Advanced Features of JViews Framework for an introduction to the use of tiling for building Web applications.

To make tiling available in the view, you must specify a tile size. The tile size is a critical parameter and must be chosen with care. See *Tile Size* in Advanced Features of JViews Framework.

```
<jvtf:equipmentView [...] tileSize="256"/>
```

Server-side caching

When the view is tiled, a server-side caching mechanism for tiles of static layers can be installed by using the `tileManager` property. No server-side caching mechanism is installed by default.

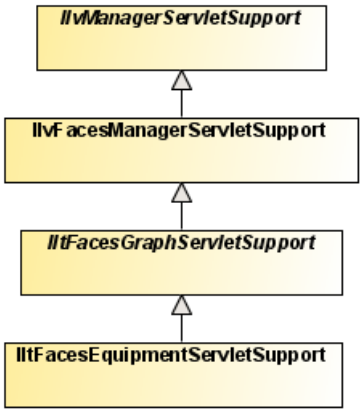
```
<jvtf:equipmentView tileManager="#{tgoBean.tileManager}" [...]/>
```

See *The Tile Manager* in Advanced Features of JViews Framework for more information.

Client-side caching

In order to enable the client-side caching mechanism, the `zoomLevels` attribute must be set. When this attribute is set, the client caches the tiles for the predefined zoom levels. See *Zoom constraints* for details on how to set the predefined zoom levels.

The API



IltFacesGraphServletSupport

The `IltFacesGraphServletSupport` determines the JViews TGO specific tiling behavior at the server-side level. By default, it uses as the static layers all the `IlvManagerLayer` instances that compose the background of your Equipment Faces component. For more information on backgrounds, see Background support.

If a custom strategy is needed for computing the tiled layers, see Developing server-side tiling which gives more information on the relevant server-side API that is need to customize the default behavior.

Managing the session expiration

The user session expires after a certain period of inactivity, usually defined in the Web deployment descriptor.

JViews objects are stored in the HTTP user session. For example, after the user session expiration, queries to update the image will fail.

The `beforeSessionExpirationHandler` property allows you to add a JavaScript™ handler that will be invoked when the user session is about to expire.

For example, to keep the session alive as long as the browser page is open, use the following code:

```
<jvtf:equipmentView [...]
    beforeSessionExpirationHandler="view.updateImage();" />
```

This example shows how to query an image and keep the user session alive.

Note the use of `view`, the implicit object that represents the view JavaScript proxy. The internal timer is reset only by requests issued by JViews objects. If the application implements other requests that do not refresh the image, this timer could be inaccurate. To reset the timer manually, use the following JavaScript code:

```
viewID.getObject().resetSessionExpirationTimer();
```

where `viewID` is the value of the `id` property of your view component.

Note: The `beforeSessionExpirationHandler` is called two minutes before the actual session expiration time.

Equipment view component services

Most of the JViews TGO equipment services can be used in the equipment faces context, with no modifications. The services that are fully compatible are:

- Interacting with the equipment objects
- Node layout
- Link layout
- Label layout
- Layers
- Background support
- Filtering
- Accepted and excluded classes
- Setting the list of origins
- Node factory
- Link factory
- Expansion strategy

For more information on these services, refer to the Equipment component services.

The following services show some differences in the equipment faces component:

- Interacting with the equipment view
- Zooming

Interacting with the equipment view

JViews TGO equipment faces interactors are declared in the JSP™ file. This is required to specify how the web browser will react to user input; some interactions being executed directly on the client side while others are submitted to and executed on the server.

View interactors cannot be purely declared in the CSS file (as is the case for the Swing equipment component). Instead, they have to be declared together with the `selectInteractor` tag. For details on how to set a specific view interactor as the listener of a `selectInteractor`, refer to *Configuring the selectInteractor*.

Zooming

Although all three zooming modes (physical zoom, logical zoom and mixed zoom) are supported in the Faces Equipment component, some thresholds are affected by some of JViews TGO Equipment Faces specific settings, like `zoomFactor`, `zoomLevels`, `minZoomLevel`, or `maxZoomLevel`.

Deploying a JViews TGO Faces application

Explains how to deploy a JViews TGO Faces web application to a Tomcat 6.0.14 servlet container, with the JavaServer™ Faces 1.2 reference implementation.

In this section

Overview
Provides basic information on how to deploy a JViews TGO Faces application based on a Tomcat 6.0 server.

JViews TGO Faces dependencies
Lists the jars required to deploy a JViews TGO Faces web application.

JViews Faces configuration at JViews Framework level
Describes the settings that are available for the JViews Faces Framework. It contains the following topics:

Web server configuration
Describes the Web server settings that users may want to take into account when deploying JViews Faces based applications.

Using JViews components with ICEfaces
Describes how to use JViews JSF components as ICEfaces components in an ICEfaces development environment.

Web application server support
Provides information on the servers to which JViews Web applications can be deployed.

Supporting Facelets and Trinidad
Explains how to make JViews Framework Faces components compatible with Facelets and Trinidad.

Overview

Like any JSF application, a JViews TGO Faces application can be deployed to any servlet container that supports the Servlet 2.3 and JSP™ 2.1 specifications. If you want to deploy web applications on a server other than Apache® Tomcat 6.0, follow the server's standard procedure.

JViews TGO Faces dependencies

The JViews TGO Faces components are a set of faces components declared in the tag library descriptor (`.tld`) file `jviews-tgo-faces.tld`, and implemented by Java™ and JavaScript™ objects. Everything is packed into the `jviews-tgo-all.jar` file. However, in order to deploy a JViews TGO Faces web application, you also need to include the following required jars in the WEB_INF/lib directory of the web application:

- `jviews-tgo-all.jar`
- `jviews-diagrammer-all.jar`
- `jviews-maps-all.jar`
- `jviews-framework-all.jar`
- `jviews-framework-thin.jar`
- `jsf-api-1.2_04-b07.jar`
- `jsf-impl-1.2_04-b07.jar`
- `jstl-1.2.jar`
- `xercesImpl-2.9.1.jar`
- `svgdom-1.0.jar`
- `commons-beanutils-1.6.jar`
- `commons-collections-2.1.jar`
- `commons-digester-1.5.jar`
- `commons-logging-1.0.4.jar`

Note: Depending on the version of the technologies used, you may need a different set of jar files. This list of files is targeted at the software configuration provided by default in the installer.

There are specific cases in which JViews TGO requires additional jar files (see JAR files for special cases). In such cases, you also need to include the required jar files in your web application.

JViews Faces configuration at JViews Framework level

Required settings

The standard configuration needed by a JSF application in the `web.xml` of your application server is shown in the following code:

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

```
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

The JViews Faces Framework needs two additional settings in order to execute correctly, namely:

- JViews Controller Servlet

The JViews Controller Servlet is in charge of loading the various resources used by the JViews Faces Framework implementation like JavaScript™ libraries, images and the like. But more importantly it provides clients with the latest state of their views capabilities as well as their dynamically generated images.

You must declare and map the JViews Controller Servlet. To do this, use the following code:

```
<servlet>
   <servlet-name>Controller</servlet-name>
   <servlet-class>ilog.views.faces.IlvFacesController</servlet-class>
   <load-on-startup> 1 </load-on-startup>
</servlet>
```

```
<servlet-mapping>
   <servlet-name>Controller</servlet-name>
   <url-pattern>/_contr/*</url-pattern>
</servlet-mapping>
```

- `ilog.views.faces.CONTROLLER_PATH`

This setting provides the users with the flexibility of defining a custom `<url-pattern>` for the JViews Controller Servlet that will be appropriately communicated to the JViews Faces Framework so that proper execution takes place.

You must set the `ilog.views.faces.CONTROLLER_PATH` context parameter which must match the content of the `<url-pattern>` of the JViews Controller Servlet without the wildcard part.

For example, the following code would appear after the code for the JViews Controller Servlet.

```
<context-param>
  <param-name>ilog.views.faces.CONTROLLER_PATH</param-name>
  <param-value>/_contr</param-value>
</context-param>
```

Optional settings

The following optional setting is available in the JViews Faces Framework: `ilog.views.faces.CONTENT_LENGTH_ENABLED`.

The `ilog.views.faces.CONTENT_LENGTH_ENABLED` setting allows users to specify whether the underling servlet that is used to generate the client-side representation of the JViews Faces Components is interacting with the client in a buffered mode or not. More specifically, it enables the communication of the content length when the server responds to client requests. This provides more optimal interaction between the client and the server.

For more insights see `javax.servlet.ServletResponse.setContentLength` and related material on the Internet.

This setting is exposed through the context parameter facility and can be set as shown in the following code:

```
<context-param>
  <param-name>ilog.views.faces.CONTENT_LENGTH_ENABLED</param-name>
  <param-value>true</param-value>
</context-param>
```

Note: Although `ilog.views.faces.CONTENT_LENGTH_ENABLED` is optional, you are recommended always to set this setting to `true`.

Web server configuration

Session persistence

Web servers often implement a session persistence mechanism used typically for traditional server clustering and fail-over techniques.

Often, the JViews Faces components are not serializable as they pertain to view related abstractions, which are typically not persistable and are stored in the HTTP session.

In order to prevent the typical serialization warnings derived from this mismatch, you can disable the session serialization mechanism for the JViews Faces based application.

How to disable session persistence in TOMCAT at web application level

1. Create a file `context.xml` and place it on the META-INF directory of your `.war` file.
2. Use the Apache™ TOMCAT configuration setting shown in the following code to disable the session serialization mechanism:

```
<Context path="/your-application-path">
   <Manager className="org.apache.catalina.session.StandardManager"
    pathname=""/>
</Context>
```

Note:
1. All the JViews Faces samples already have this session serialization setting disabled for TOMCAT at this level.
2. These settings apply to TOMCAT 6.0.14 and later.

How to disable session persistence in TOMCAT at web server level

1. Modify the TOMCAT/conf/context.xml to use this as the Session Manager definition:

```
<Manager pathname=""/>
```

Note: These settings apply to TOMCAT 6.0.14 and later.

For more details on these settings see the TOMCAT configuration documentation.

For details on how to disable session serialization with your Web server, refer to its configuration documentation.

Running JViews faces components in JSR 168 portlets

Note: See the Release Notes for supported JSF implementations and JSF Portlet bridge combinations.

If you want to use JViews Framework Faces components in a JSR 168 portlet environment, you first need to check with your portal vendor whether JavaServer™ Faces components are supported.

Your Web application must be correctly configured. This section describes each of the steps required to make JViews Framework Faces components compatible with portlets.

Note: JViews Framework Faces components are automatically switched to portlet mode if the classes of the portlet API are detected in the class path.

To avoid naming clashes between portlets, the JSR 168 specification requires content to be generated that is unique to each portlet. Therefore, the generated variables used by JViews Framework Faces components must be prefixed by the portlet namespace.

Scripts prefixed by a namespace

From JViews 8.5, the servlet filter `ilog.views.faces.servlet.IlvJSNamespaceFilter` is no longer needed and must not be set on the controller servlet.

JavaScript variables prefixed by a namespace

In portlet mode, the generated JavaScript™ variables are prefixed by the portlet namespace. Thus, their usage in the JSP™ page is quite different.

A JavaScript action is built on a managed bean by using the `IlvFacesUtil.encodeJavaScriptVariables(String)` static method.

The parameter is the desired JavaScript action where the variables are declared with the `${id}` notation. For example:

```
IlvFacesUtil.encodeJavaScriptVariables("${view}.setInteractor(${inte
ractor})");
```

where `view` and `interactor` represent JavaScript variables.

The JViews Faces components that have JavaScript handlers need only to reference these bean properties.

The following code shows a more complete use of JavaScript actions in the JSP page and the managed bean.

Example 1:

```
[...]
<jvf:zoomInteractor id="zoom" />
<jv:imageButton onclick="#{frameworkBean.setZoomAction}"/>
<jvf:view id="view" />
[...]
```

Example 2:

```
public class FrameworkBean {
[...]
private String setZoomAction;
public FrameworkBean(){
setZoomAction =
IlvFacesUtil.encodeJavaScriptVariables("${view}.setInteractor(${zoom})");
}
public String getSetZoomAction(){
return setZoomAction;
}
[...]
}
```

Declaring the image servlet

In portlet mode, the servlet used to render the image must be declared:

```
<jvf view [...] servlet=
"ilog.views.faces.dhtml.servlet.IlvFacesManagerServlet />"
```

Integrating JSF components into the portal

Depending on your portal implementation, integrating JSF components may require special configuration that is conditioned by the application server, the JSF implementation, the portlet-JSF bridge, and so on. Check with your portal vendor for what you need to do in this configuration step.

Using JViews components with ICEfaces

Settings for using JViews components in ICEfaces

This section describes the settings you need to use JViews JSF components with ICEfaces.You are assumed to be familiar with Web application development using JSF technologies. You need to have JViews 8.5 or above and ICEfaces 1.7.2 or above installed. You can go to *http://www.icefaces.org* to download a more recent version of ICEfaces. If you use Eclipse® , ICEfaces also has a plug-in for this environment.

Since JViews 8.5, JViews JSF components support ICEfaces completely. JViews requires the standard request mode of ICEfaces. This is the mode in which ICEfaces interoperates with third-party components. To set this mode, you need to add the following element to the `web.xml` file of your Web application.

```
<context-param>
   <param-name>com.icesoft.faces.standardRequestScope</param-name>
   <param-value>true</param-value>
</context-param>
```

For other settings required by JViews JSF components, see *Required settings*.

For more settings and concrete examples, look at the sample installed in ***<installdir>/samples/faces/jsf-tgo-ice***.

Interoperability between JViews components and ICEfaces components

This section describes the interoperability between JViews components and ICEfaces components. JViews components and ICEfaces components are both JSF components. They can work together both on the client side and on the server side.

On the client side, JViews JSF components are high-level Ajax-enabled JavaScript™ objects. You can direct the behavior of JViews components by invoking their JavaScript methods. For example, when you click an ICEfaces button you can update the contents of a JViews view by calling its JavaScript method: `updateImage()`.

On the server side, both JViews components and ICEfaces components can be bound to backing beans. This allows you to exchange parameters and data between the backing beans of JViews components and ICEfaces components.

The interoperability between JViews and ICEfaces components can involve both the server side and the client side.

Suppose that you have a diagram view showing a number of nodes and links. You want to display a particular node and center it on the screen when you click an ICEfaces button. This use case is shown in the code example at ***<installdir>/../jviews-diagrammer86/codefragments/jsf-diagrammer-ice/srchtml/diagrammer.jsp.html***

Run this sample now to understand the situation better.

The action is initiated on the client side by clicking a button. However, the task cannot be performed completely on the client side because there is not enough information on the

selected node. Therefore you have to submit the request to the server and ask the server to perform more computation.

Once the backing bean on the server side has computed the offset to be applied to center the selected node on the screen, you need to find a way to tell the client-side JViews components to apply that offset. For this purpose, ICEfaces provides a way for you to send JavaScript code from the server to the client. The code is as follows.

```
com.icesoft.faces.context.effects.JavascriptContext
   .addJavascriptCall(FacesContext.getCurrentInstance(),
   "diagrammer.moveTo(300, 500);");
```

The ICEfaces Ajax agent on the client will evaluate the received JavaScript code in order to scroll the diagram to the expected position.

For more details, see the `DiagrammerBean.java` file in the same sample.

Push updates to JViews components

This section describes the techniques for push updates (server-initiated rendering) with JViews components. One of the interesting features of ICEfaces is its server-initiated rendering. This technique allows push updates to components rendered by Web browsers. This topic explains how to make push updates to JViews components.

JViews components are Ajax-enabled components and their contents are generally GIF or PNG images generated by JViews server-side servlet supports. There is no way to push images directly to JViews components.

ICEfaces is able to push things such as HTML fragments and JavaScript code but not images. However, you can use the ICEfaces push mechanism to notify client-side JViews components that updates are available on the server. Then the JViews components can use the Ajax mechanism to get the updated images. This approach is quite efficient in terms of network traffic.

To notify client-side JViews components, you can use the ICEfaces server-initiated rendering technique to push JavaScript code. The ICEfaces Ajax agent will receive and evaluate the code. For example, you can put something like the following in JavaScript code:

```
<script type="text/javascript">diagrammer.updateImage();</script>
```

This code tells a JViews diagram component to update its contents.

For tips and tricks on how to push JViews components, look at the push example installed with JViews TGO in ***<installdir>/../jviews-diagrammer86/codefragments/jsf-diagrammer-ice***.

ICEfaces software in JViews

This section describes the ICEfaces binary files provided with JViews and lists the known issues.

ICEfaces binary files provided with JViews

ICEfaces binary files are included in the JViews distribution so that the integration code samples can run out-of-the-box. ICEfaces jar files can be found under `/lib/external`. However, the full ICEfaces distribution is not included.

To get a complete or more updated distribution, you can get ICEfaces under the Mozilla® Open Source License 1.1. ICEfaces source code is available at *http://www.icefaces.org*. A copy of the Mozilla Open Source License 1.1 is available there or at *http://www.mozilla.org/MPL/MPL-1.1.htm*.

Known ICEfaces issues

The following issue exists when using ICEfaces components with JViews components:

- ICEfaces is not able to parse JViews component `<jvf:view>` in JSP mode probably because it confuses this tag with `<f:view>`, although they are in different namespaces. A workaround has been found. See the Graphic Framework example and the `iview.tld` file in the code example installed in ***<installdir>/../jviews-diagrammer86/codefragments/jsf-diagrammer-ice***.

Web application server support

Apache® Tomcat 6.0.14 is the reference Web application server shipped with product_name. Other Web application servers such as JBoss AS 4.2.3.GA, IBM® WebSphere® 7.0, and Oracle® Weblogic Server 10.3 have also been tested.

You may need related infomation when deploying JViews Web applications to servers as follows:

JBoss Application Server 4.2.3.GA

JBoss AS 4.2.3.GA has a JSF implementation included. To avoid conflicts, you should not include JSF jars in your `war` file when deploying JViews Web applications.

When deploying JViews Facelets Web applications, you may need to exclude `dom-3.0.jar` from the `war` file to avoid XML parsing exceptions.

JBoss AS 4.2.3.GA does not support multi-pattern <servlet-mapping> elements in `web.xml`. You must use multi <servlet-mapping> elements with separated patterns.

IBM WebSphere 7.0

WebSphere 7.0 has a JSF implementation included. To avoid conflicts, you should not include JSF jars in your `war` file when deploying JViews Web applications.

When deploying JViews Facelets Web applications, you may need to exclude `dom-3.0.jar` from the `war` file to avoid XML parsing exceptions.

There is a known issue when deploying ICEfaces applications to WebSphere, see *http://jira.icefaces.org/browse/ICE-2330*.

Oracle WebLogic Server 10.3

You need to change the schema of your `web.xml` file to 2.5.

For the exception that the deferred EL expression is not allowed since `deferredSyntaxAllowedAsLiteral` is false, you need to add `<%@ page deferredSyntaxAllowedAsLiteral="true" %>` in the JSP page.

In the Trinidad and Facelets samples, the TGO network view might not be shown. You need to move the interactors out of the `tr:panelTabbed` component.

For Trinidad samples with invalid PPR responses, the problem is caused by an invalid XML response, which has been reported in *https://issues.apache.org/jira/browse/TRINIDAD-1170*.

Supporting Facelets and Trinidad

If you want to use JViews Framework Faces components in a Facelets context, your Web application must be correctly configured.

Compatibility with Facelets and Trinidad

To make JViews Framework Faces components compatible with Facelets and Trinidad:

- Edit the configuration files.

  For complete application samples configured for use with Facelets or Trinidad, see ***<install-dir>* /samples/faces/jsf-tgo-facelets**.

  To see examples of correct settings for Facelets with Trinidad, look at the `faces-config.xml` and `web.xml` files. If you want to use Facelets without Trinidad, look at `faces-config-std.xml` and `web-std.xml` instead.

- Develop XHTML-based pages according to the tag library documentation.

  All attributes and all tags except the menu tags listed in *Contextual menus* are supported in Facelets.

  If you are using custom tags, make sure you provide a `custom.taglib.xml` file that describes your custom library and declare its XML namespace in the page.

- Make sure that your `.war` files (or your server default libraries) include the necessary Facelets (and possibly Trinidad) jar files.

Contextual menus

In a Facelets context, you will be able to provide dynamic menus through the `factory` or `factoryClass` attribute of a contextual menu object but you will not be able to use `menu`, `menuItem`, or `menuSeparator` tag components directly in the page.

```
<... contextualMenu ...
factoryClass="mydemo.somepackage.MenuFactory" />
```

Static menu

You will be able to bind a static menu (running the code of the factory only once), in addition to dynamic menus, using the `value` attribute of the contextual menu element.

```
<... contextualMenu ... value="#{chartBean.menu}" />
```

See also Guide to using JViews components with ICEfaces.

IBM® ILOG® JViews TGO Faces technical overview

Describes the architecture of the Faces library, how requests are processed by the equipment and network faces components, and which types of interaction these components offer.

In this section

The graph architecture
Describes the graph faces architecture.

The network faces component architecture
Describes the network faces component architecture.

The equipment faces component architecture
Describes the equipment faces component architecture.

Processing requests
Explains how requests are processed by the IlpNetwork and IlpEquipment components.

Interactions
Describes the different types of interaction.

The graph architecture

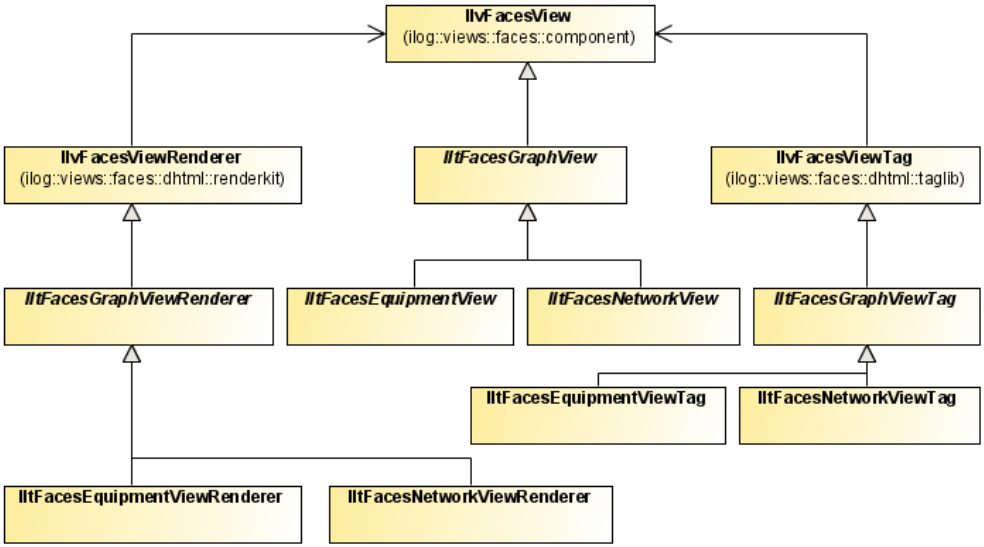
The IBM® ILOG® JViews TGO Faces library is a set of JavaServer™ Faces components that allow you to display and interact with business objects and data in the following formats:

- A network of nodes: the network faces component.
- Items of equipment composed of cards, ports and LEDs: the equipment faces component.

Like the `IlpNetwork` and `IlpEquipment` Swing components, the network and equipment faces components share the same architectural design. In a high-level abstraction, the network and equipment faces components play the role of the view (`IlpNetworkView` and `IlpEquipmentView`) as they are responsible for displaying the graphic representation of the model on the client screen.

Like the `IlpNetworkView` and `IlpEquipmentView`, which are based on `IlpGraphView`, the network and equipment faces components are based on an abstract Graph faces component which cannot be directly used in a JSP™ file. This Graph component is defined by a component abstract implementation, `IltFacesGraphView`, an abstract renderer, `IltFacesGraphViewRenderer`, and an abstract tag implementation, `IltFacesGraphViewTag`.

The Graph component is based on the `view` faces component declared in the JViews Framework Faces library. It inherits all the features and characteristics of the `view` faces component. In addition, it extends its functionality to display the specific JViews TGO business objects by using an underlying `IlpGraphView` instead of a generic `IlvManagerView`. The following class diagram shows these dependencies:



Graph Faces Architecture

The network faces component architecture

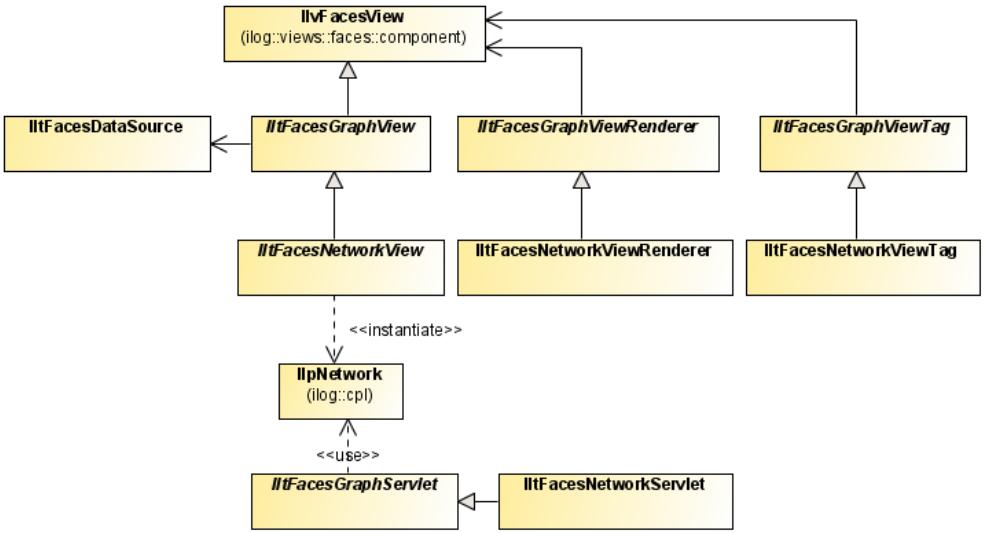
The network faces component is designed to display snapshot images of the visible area of an underlying `IlpNetwork` component. It works in conjunction with a dedicated servlet responsible for rendering static images of an `IlpNetworkView`. The component by itself is not able to process any user interaction; it must be connected to other faces components, like the select interactor (`selectInteractor`), in order to convert client-side interactions into server-side events.

Class overview

The network faces component is declared in the tag library descriptor (`.tld`) file as `networkView`. Like any faces component, it has a tag implementation, a component implementation, and a DHTML renderer; in addition, the network faces component has a dedicated servlet to handle image requests. The classes are as follows:

- `IltFacesNetworkViewTag`: The tag implementation
- `IltFacesNetworkView`: The faces component implementation
- `IltFacesNetworkViewRenderer`: The renderer for the component
- `IltFacesNetworkServlet`: A dedicated servlet to produce images

The tag implementation handles the various tag attributes declared in the tag library descriptor file for the `networkView` component. The component implementation handles all this information so that the renderer can create a DHTML representation of it while the image servlet processes the client requests, directly interacting with the underlying `IlpNetwork` component and modules through its API. The following class diagram shows these dependencies:



Network Faces Architecture

The equipment faces component architecture

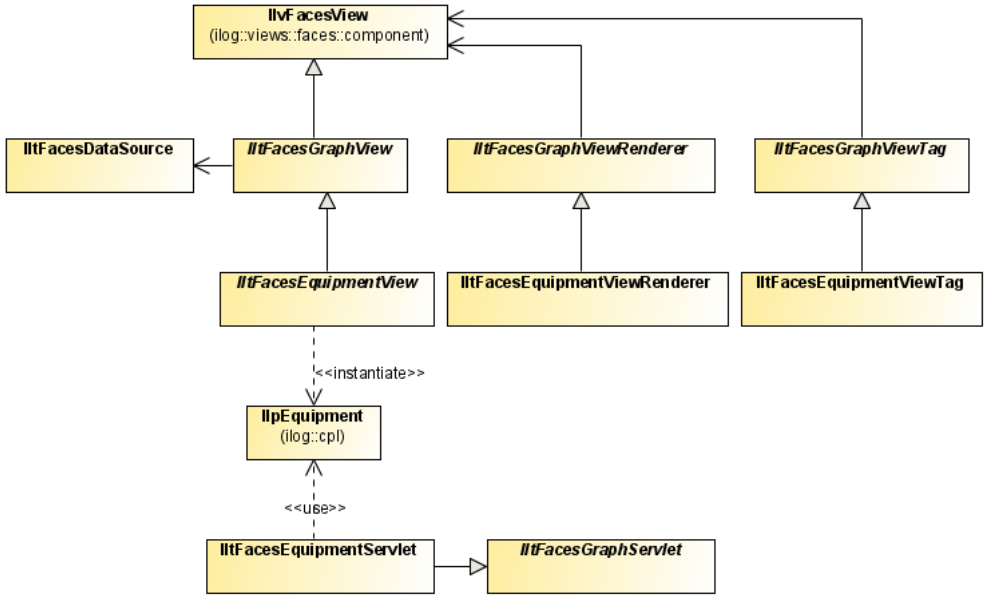
The equipment faces component is designed to display snapshot images of the visible area of an underlying `IlpEquipment` component. It works in conjunction with a dedicated servlet responsible for rendering static images of an `IlpEquipmentView`. The component by itself is not able to process any user interaction; it must be connected to other faces components, like the select interactor (`selectInteractor`), in order to convert client-side interactions into server-side events.

Class overview

The equipment faces component is declared in the tag library descriptor (`.tld`) file as `equipmentView`. Like any faces component, it has a tag implementation, a component implementation, and a DHTML renderer; in addition, the equipment faces component has a dedicated servlet to handle image requests. The classes are as follows:

- `IltFacesEquipmentViewTag`: The tag implementation
- `IltFacesEquipmentView`: The faces component implementation
- `IltFacesEquipmentViewRenderer`: The renderer for the component
- `IltFacesEquipmentServlet`: A dedicated servlet to produce images

The tag implementation handles the various tag attributes declared in the tag library descriptor file for the `equipmentView` component. The component implementation handles all this information so that the renderer can create a DHTML representation of it while the image servlet processes the client requests, directly interacting with the underlying `IlpEquipment` component and modules through its API. The following class diagram shows these dependencies:

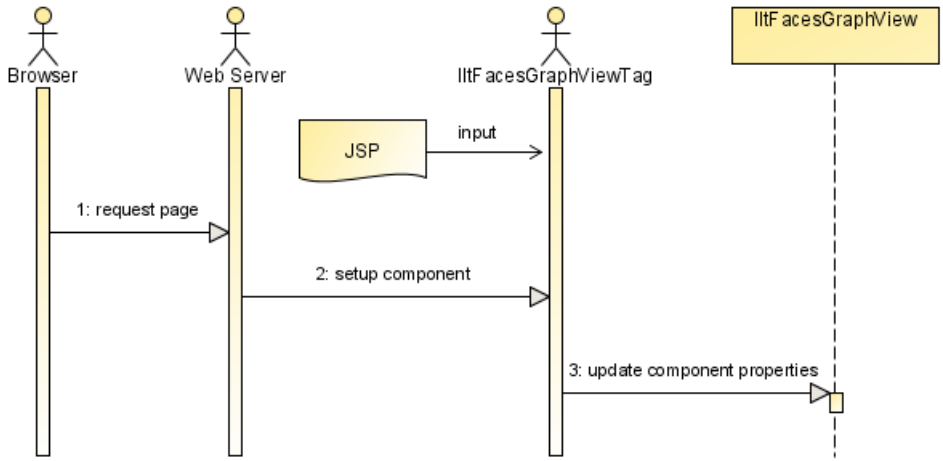


Equipment Faces Architecture

Processing requests

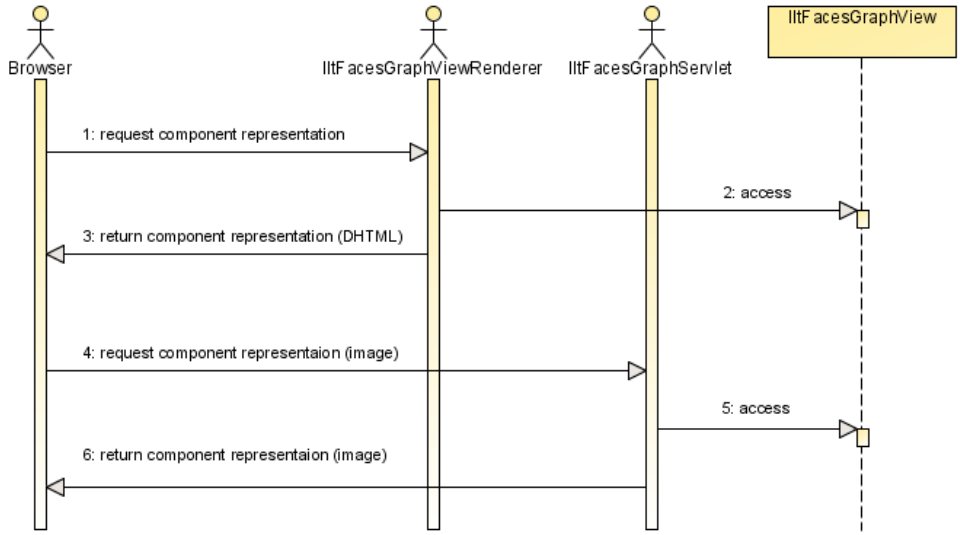
The `IltFacesNetworkView` and `IltFacesEquipmentView` faces components internally instantiate `IlpNetwork` and `IlpEquipment` components, respectively, that are responsible for handling all business data and their graphic aspects.

When a JSP™ page containing a `networkView` or an `equipmentView` component is first processed, the tag implementation, `IltFacesNetworkViewTag` or `IltFacesEquipmentViewTag`, interprets all the tag attributes and stores this information in the `UIComponent` (`IltFacesNetworkView` or `IltFacesEquipmentView`). The following diagram illustrates this:



Processing of a JSP Page

The JSF lifecycle calls the component renderer (`IltFacesNetworkViewRenderer` or `IltFacesEquipmentViewRenderer`) to encode the page into HTML. This is done by adding DHTML code into the response. Back to the client, the DHTML code is executed and a new request is sent to the server. This time the request is directed to the image servlet (`IltFacesNetworkServlet` or `IltFacesEquipmentServlet`). The image servlet generates a static snapshot of the visible area of the underlying `IlpNetworkView` or `IlpEquipmentView` and sends it back to the client to be displayed. The following diagram illustrates this:



Rendering of the Faces Components

Interactions

There are three types of interaction:

- Client-side: no roundtrip to the server
- JSF lifecycle: the interaction is processed by the JSF lifecycle, and another roundtrip is necessary to update the image from the image servlet
- Image servlet: the interaction is processed by the image servlet and the image is updated in one single roundtrip.

Basic interactions like panning the view are processed locally on the client side. They are fast and no requests are sent to the servlets.

When using advanced interactors like the select interactor (`selectInteractor`), you can choose the invocation context, or how the submitted request will be processed. There are two options:

- `JSF_CONTEXT`: interaction is processed by the JSF lifecycle
- `IMAGE_SERVLET_CONTEXT`: interaction is processed directly by the image servlet

When the JSF lifecycle processes a request, it follows well-defined phases, respecting listeners, triggering actions and notifying components. The response forces all the pages to be updated at the end, and the network or equipment component executes the DHTML code to request a new image from the image servlet. At least two roundtrips to the server take place.

When the request goes straight to the image servlet (`IMAGE_SERVLET_CONTEXT`), the processing is faster, as only the `networkView` or `equipmentView` component is updated. The response carries the new image in one single roundtrip to the server. The drawbacks are that no other faces component is updated, and that the results may be inconsistent.

Note: Unlike the other faces components, the `overview` faces component is always updated to appropriately display the latest state of the view, regardless of which option is used by the interactor (JSF_CONTEXT or IMAGE_SERVLET_CONTEXT).

Index