IBM ILOG JViews Maps V8.6

Building Web Applications

Copyright notices

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.

IBM ILOG JViews Maps copyright

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

Table of contents

Introducing the Web technologies used in JViews Maps

This document provides information on how to deploy your application as an Internet-based application. It discusses the two major categories of Internet applications: thin client applications and rich Web applications.

In this section

Overview
Gives an overview of Internet-based applications.

Thin client applications
Describes thin client applications and use of JViews Faces components.

Rich Web applications
Introduces rich Web applications.

Overview

The versatility of Java™ deployment was one of the key factors driving the adoption of Java. For many years, Java has been recognized for its multiplatform capabilities, for example, running on both Microsoft® Windows® and Linux® . Java covers a wide spectrum of execution environments, from traditional desktop environments to Internet-based applications.

Thin client applications

Describes thin client applications and use of JViews Faces components.

In this section

Thin client application designs
Gives an overview of what thin client applications are.

Ajax-enabled components
Describes the use of Ajax-enabled JViews components in Web applications.

Thin client application designs

As their name implies, thin client applications deploy minimal code on the clients and rely heavily on the server to deal with user interactions and to respond with corresponding displays.

In such application designs, application deployment is transparent and updates are immediately available to all users. Application management can be centralized on a few localized servers. Thus it requires fewer administration resources and helps to maximize the availability of the application.

Against these advantages, you must weigh the most common drawbacks, which are:

- The relatively slow reaction of the application to user input.
- Poor server scalability for handling a large user base.
- Poor to no offline capability.
- Lack of advanced interactive graphics: since the local processing power is not leveraged, the user's machine is used only to display Web pages.

JViews Maps provides advanced capabilities for such application designs. It relies on JavaServer™ Faces (JSF) as the server-side component model and Dynamic HTML (DHTML) as the client-side display technology. This combination facilitates development work and provides easier integration with third-party components and tools.

Going beyond simple thin clients, JViews Maps thin client leverages the local execution capabilities of JavaScript™ to provide an advanced user experience; for demanding interactions, Asynchronous JavaScript And XML concepts, or Ajax, are applied.

Ajax-enabled components

With JViews Maps Faces components and JavaScript™ you can develop a new generation of highly responsive, highly interactive Web applications. The high responsiveness is achievable through Ajax, which supports asynchronous and partial refreshes of a Web page. A partial refresh means that when an interaction event fires, a Web server processes the information and returns a response specific to the data it receives. The server does not send back an entire page to the client of the Web application.

Why asynchronous? The client can continue processing while the server processes in the background. A user can continue interacting with the client without noticing latency in the response. The client does not have to wait for a response from the server before continuing, as in the traditional synchronous approach.

See *Using DHTML-based JSF Components to build Web applications* and *Deploying a JViews Maps application as a DHTML-only thin client* for more information about these deployment strategies.

Rich Web applications

Introduces rich Web applications.

In this section

Overview
Gives an overview of what rich Web applications are.

Applets
Introduces applets.

Java Web Start applications
Introduces Java™ Web Start applications.

Overview

In the last few years, rendering technologies such as Flash® or Scalable Vector Graphics (SVG) have emerged to overcome some user interaction issues and display limitations found with the DHTML rendering described in *Thin client application designs*. In parallel, the role of the client has been promoted to further leverage local processing power through JavaScript™ . The objective is to improve user experience on the client and scalability on the server and has led to the Ajax concept.

In such designs, servers are partially offloaded to focus mainly on data handling and less on screen generation.

JViews Maps helps you to develop such applications as:

- Applets
- Java™ Web Start Applications

Applets

An applet is a traditional Java™ application that is wrapped as an applet and automatically transferred by the server as needed.

Thus it retains the advantages of the thin client, provides more advanced user interactions, and minimizes the server workload. The main drawbacks are that a Java virtual machine needs to be installed in each execution environment and initial loading time can be long and stressful for networks, since applications can be many megabytes.

When developing a JViews Maps application using this approach, see Programming with JViews Maps.

Java Web Start applications

Like applets, Java™ Web Start applications allow for traditional development techniques, but applications have off-line capabilities and are cached locally in the execution environment.

They minimize start-up time and network bandwidth requirements, since servers only distribute an application when updates are available. The major known drawback is the need to install a Java Web Start environment that can be transparently streamed, but is sometimes blocked by some network security policies.

When developing a JViews Maps application using this approach, see Programming with JViews Maps and the Java Web Start documentation at *http://java.sun.com/products/javawebstart/developers.html*.

Using DHTML-based JSF Components to build Web applications

Shows how to use the components of IBM® ILOG® JViews Maps Faces to create JavaServer™ Pages (JSP™) compliant with JavaServer Faces (JSF).

In this section

Overview
Shows you how to use the components of JViews Maps Faces to create JavaServer™ Pages compliant with JavaServer Faces.

The architecture of JViews Maps Faces
Presents an overview of the architecture of JViews Maps Faces.

The JViews Maps Faces component set
Illustrates how to use the Faces components of JViews Maps.

JavaScript objects
Describes the use of JavaScript™ objects.

Contexts for actions on the view

Integrating JViews Faces in your environment
Provides information about configuring a JSF application in the application server, session persistence, JSR 168 portlets, ICEfaces, and Facelets and Trinidad.

Overview

This section shows you how to use the components of JViews Maps Faces to create JavaServer™ Pages (JSP) compliant with JavaServer Faces (JSF). JViews Maps Faces Components are available as a set of classes and a tag library. A set of renderers generate DHTML code for rendering the components. The components also use servlet technology to generate images to be transferred to the client.

JViews Maps Faces provide Ajax-enabled components for developing highly responsive and interactive Web applications.

The architecture of JViews Maps Faces

Presents an overview of the architecture of JViews Maps Faces.

In this section

About support for JViews Faces
Describes thin-client support based on JavaServer™ Faces (JSF) technology.

Servlet and component classes
Identifies servlet and component classes for generating the visual representation of the component.

About support for JViews Faces

The thin-client support based on JavaServer™ Faces (JSF) support consists of:

- The tag library (a set of JSP™ tags)
- A Java™ API
- A set of DHTML objects

The JSP™ tags are used to build JSP pages. Each tag represents a component and has a set of attributes for configuring the component.

Not all the components have a visual representation. For example, an interactor is intended only to be set on a view and has no visual representation.

When a tag is processed by the JSP engine, it is compiled into Java code that is executed to produce the page content. The tag library produces DHTML objects. Each object can be referenced by JavaScript™ code and can be modified on the client side without a server roundtrip.

See the Release Notes for the Web browsers and versions with which the JViews Maps Faces components are compatible.

Component set of JViews Maps Faces

The JViews Maps Faces component set includes:

- A view
- A legend tool

Component set of JViews Diagrammer Faces

The JViews Diagrammer Faces component set includes:

- A view
- An overview
- A pan tool
- A zoom tool
- A set of interactors
- A popup menu

Component set of JViews Framework Faces

The JViews Framework Faces component set includes:

- A view

- An overview
- A pan tool
- A zoom tool
- A set of interactors

Servlet and component classes

JSF components in JViews Maps use servlet technology to produce the images that are the visual representation of the component on the client side. Dedicated servlet, servlet support, and components are available to help create an application.

Servlet and component classes in JViews Maps

Name	Description
In package `ilog.views.maps.servlet`: `IlvFacesMapsServlet`	A dedicated `IlvFacesDiagrammerServlet`.
In package `ilog.views.maps.servlet`: `IlvFacesMapsServletSupport`	A dedicated `IlvFacesDiagrammerServletSupport`.
In package `ilog.views.maps.servlet`: `IlvFacesGoogleViewServlet`	A dedicated `IlvFacesMapsServlet`
In package `ilog.views.maps.servlet`: `IlvFacesGoogleViewServletSupport`	A dedicated `IlvFacesMapsServletSupport`
In package `ilog.views.maps.faces.dhtml.component`: `IlvFacesDHTMLMapView`	A diagram view with additional support for maps. A dedicated `IlvFacesDHTMLDiagrammerView`
In package `ilog.views.maps.faces.dhtml.component`: `IlvFacesGoogleViewComponent`	A map view overlayed on Google Maps™ thin client.
In package `ilog.views.maps.faces.component`: `IlvFacesLayerVisibilityTool`	The layer tool is a display tool that allows the user to change the visible layers.

Servlet and component classes in JViews Diagrammer

Name	Description
ilog.views.diagrammer.faces.dhtml.servlet. `IlvFacesDiagrammerServlet`	A dedicated `IlvDiagrammerServlet`.
ilog.views.diagrammer.faces.dhtml.servlet. `IlvFacesDiagrammerServletSupport`	A dedicated `IlvDiagrammerServletSupport`.
ilog.views.diagrammer.faces.dhtml.component. `IlvFacesDHTMLDiagrammerView`	A diagram view component extended to have DHTML rendering.
ilog.views.diagrammer.faces.dhtml.interactor. `IlvFacesNodeOrLinkSelectInteractor`	An interactor that allows you to select a node or a link in the JSF context by clicking the image.
ilog.views.diagrammer.faces.dhtml.interactor. `IlvFacesNodeOrLinkSelectRectInteractor`	An interactor that allows you to select nodes and links in the JSF context by dragging a rectangle on the image.
ilog.views.diagrammer.faces.dhtml.interactor. `IlvFacesSelectInteractor`	An interactor that allows you to select and move nodes and links in the JSF context.
ilog.views.diagrammer.faces.dhtml.component. `IlvFacesDiagrammerSelectionManager`	A component that allows users to configure how selection management on the `IlvFacesDHTMLDiagrammerView` works.

The diagram component also uses the following component classes of the JViews Framework:

- `IlvFacesDHTMLOverview`

- `IlvFacesZoomTool`
- `IlvFacesPanTool`
- `IlvFacesPanInteractor`
- `IlvFacesZoomInteractor`
- `IlvFacesContextualMenu`

Servlet and component classes in JViews Framework

Name	Description
`IlvFacesManagerServlet`	A dedicated `IlvManagerServlet`.
`IlvFacesManagerServletSupport`	A dedicated `IlvManagerServletSupport`.
`IlvFacesDHTMLView`	A view component extended to have DHTML rendering.
`IlvFacesDHTMLOverview`	An overview component.
ilog.views.faces.component. `IlvFacesZoomTool`	A tool that allows you to choose a view zoom level.
ilog.views.faces.component. `IlvFacesPanTool`	A tool that allows you to pan in each direction and to fit the view.
ilog.views.faces.interactor. `IlvFacesZoomInteractor`	An interactor that allows you to zoom the view.
ilog.views.faces.interactor. `IlvFacesPanInteractor`	An interactor that allows you to pan the view.
ilog.views.faces.interactor. `IlvFacesMapInteractor`	An interactor that allows you to execute an action in the servlet context by clicking the image.
ilog.views.faces.interactor. `IlvFacesMapRectInteractor`	An interactor that allows you to execute an action in the servlet context by dragging a rectangle on the image.
ilog.views.faces.dhtml.interactor. `IlvFacesObjectSelectInteractor`	An interactor that allows you to execute an action in the JSF context by clicking the image.
ilog.views.faces.dhtml.interactor. `IlvFacesObjectSelectRectInteractor`	An interactor that allows you to execute an action in the JSF context by dragging a rectangle on the image.
ilog.views.faces.component. `IlvFacesContextualMenu`	A contextual popup menu.

The JViews Maps Faces component set

Illustrates how to use the Faces components of JViews Maps.

In this section

Overview
Shows the class relationships of the Faces component set.

Creating simple views
Explains how to create various types of simple view.

Creating a Google Maps view
Explains how to use the JViews Maps Faces Google™ View component to display a Google™ Map in a page.

Controlling the display of the view
Explains how to control the visible area of a view and how to reset or change it.

Zoom constraints
Explains zoom factors and the constraints on zooming and panning.

Zoom levels and dynamic layers
Explains fixed and dynamic zoom layers.

Tiling the view
Explains how to set a tile size and other attributes in order to use tiling in a view.

Visible layers
Describes how to use the `visibleLayers` property to specify the manager layers displayed by a view.

Image maps
Explains how to add an image map to an image on the client side.

Adding a popup menu
Explains how to add a popup menu.

Styling the popup menu
Explains how to use CSS classes to set popup menu properties for styling purposes.

Adding a legend
Explains how to add a legend.

Adding a Message Box
Shows how to connect a message box to your diagram.

Adding an Overview
Shows how to add an overview to display the global area and a rectangle that corresponds to the visible area in the main view.

Adding a Pan Tool and a Zoom Tool
Shows how to add a pan tool or zoom tool to the view to allow panning and zooming.

Server-side caching
Describes how to use server-side caching with a tiled view.

Managing the session expiration
Describes the implications of session expiration and how to keep a user session alive when it is about to expire.

The map view as a Diagrammer view
Presents the map view as a Diagrammer view.

JViews Maps project
Shows how to configure the content of a map view.

JViews Diagrammer Designer Project
Shows how to configure the style and data source of a diagram component by setting a Designer project.

Data Source Binding
Shows how to connect a data source component to the diagram.

Styling with CSS
Shows how to customize the way the data source is displayed using Cascading Style Sheets (CSS).

Installing Interactors
Shows how to install an interactor on the view to interact with it.

Select Interactor
Describes the select interactor, used to select one or more objects and move them without performing a full screen refresh.

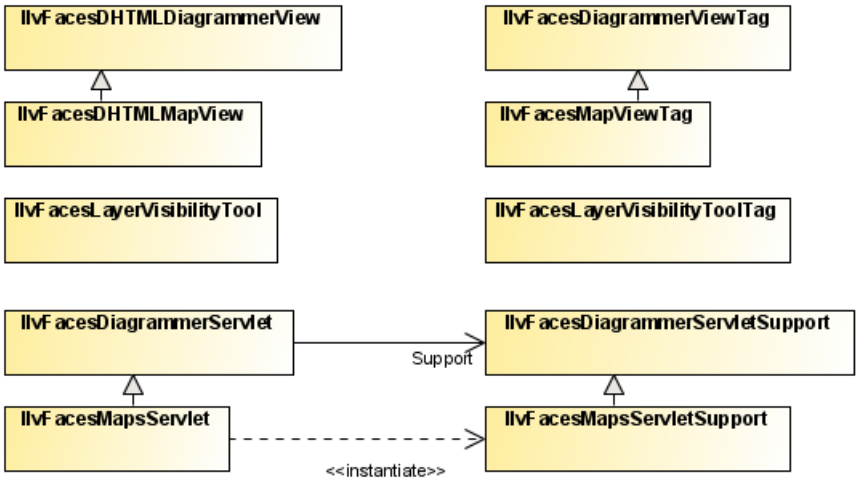
Creating nodes and links
Shows how to create nodes and links on the view.

Deleting selected nodes and links
Shows how to delete selected nodes and links on the view.

Dashboard diagram
Shows how to add a dashboard component to the view.

Overview

The class relationship of the JViews Maps Faces component set is shown in *JViews Maps Faces Component Set UML Diagram*.



JViews Maps Faces Component Set UML Diagram

Creating simple views

The view component is the central component of a JViews Faces application. All the other components depend on or interact with this view. The first and simplest page that can be made with a JViews Faces component is an empty view.

Creating a map view

The map view extends the diagram view, so the common overview, zoom tool, pan tool, and interactor components are compatible.

Creating an empty view

To specify an empty view:

```
<jvmf:mapView style="width:500px; height:300px;" />
```

This produces a 500 by 300 pixel view.

Declaring the namespace

The namespace `jvmf` (for JViews Maps Faces) must be declared in the page.

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

Using the width and height attributes

Using the style to specify the size of the component is preferable, but an alternative is to use the `width` and `height` attributes.

```
<jvmf:mapView width="500" height="500" />
```

Creating a diagram view

The diagram view extends the common view so the common overview, zoom tool, pan tool, and interactor components are compatible.

Creating an empty view

To specify an empty view:

```
 <jvdf:diagrammerView style="width:500px; height:300px;" />
```

This produces a 500 by 300 pixel view.

Declaring the namespace

The namespace `jvdf` (for JViews Diagrammer Faces) must be declared in the page as follows:

```
<%@ taglib
```

```
    uri="http://www.ilog.com/jviews/tlds/jviews-diagrammer-faces.tld"
    prefix="jvdf" %>
```

Using the width and height attributes

Using the style to specify the size of the component is preferable, but an alternative is to use the `width` and `height` attributes.

```
<jvdf:diagrammerView width="500" height="500" />
```

Creating a view at the JViews Framework level

This view can be extended to make more specific components.

Creating an empty view

Specify an empty view.

```
<jvf:view style="width:500px; height:300px;" />
```

This produces a 500 by 300 pixel view.

Declaring the namespace

The namespace `jvf` (for JViews Framework Faces) must be declared in the page as follows:

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

Using the width and height attributes

Using the style to specify the size of the component is preferable, but an alternative is to use the `width` and `height` attributes.

```
<jvf:view width="500" height="500" />
```

Creating a Google Maps view

A simple page that can be made with the JViews Maps Faces Google™ View component is to display a Google™ Map in the page.

Creating a simple Google Maps view

```
<jvmf:googleView style="width:500 px; height:300 px;" level="6" lon="-100"
lat="40"
key="myGoogleMapKey"/>
```

This produces a 500 by 300 pixel view, centered on 40°N 100°W, at a zoom level of 7.

Declaring the namespace

The namespace jvmf (for JViews Maps Faces) must be declared in the page:

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

The Google View component makes use of the Google Maps API (client side). The Google key provided in the component should contain a valid application key in order to allow the component to display on any server (apart from the localhost). To retrieve that key, you need to access *http://code.google.com/apis/maps/signup.html*, “Sign Up for the Google Maps API” and register your key before using the JViews Google Maps component.

An alternative to specifying the size of the component is to use the width and height attributes, but using the style is preferable.

Using the Width and Height attributes

```
<jvmf:googleView width="500" height="300" level="6" lon="-100" lat="40"
key="myGoogleMapKey"/>
```

Note: In some environments, for example those that use PPR (Partial Page Refresh), you may have to register the key in a separate statement, and not use it in the component tag itself.

Pre-declaring the Google Maps key in trinidad

For example, in trinidad:

```
<trh:script source="http://maps.google.com/maps?file=api&v=2&2=mykey" />
...
<tr:showDetailItem text="Background map">
<jvmf:googleView id="mapID" level="7" style="width:500px;height:300px" data="/
data/world.ivl" />
</tr:showDetailItem>
```

The overview, zoom tool, pan tool, and interactors common component are not compatible with this view, but are provided through the use of Google Maps Controls. You can select the controls to display with the `controls` attribute (see *http://code.google.com/apis/maps/documentation/controls.html*).

The main use of this JSF component is to overlay JViews map layers, symbols and links on a Google Maps background. This can be done through the `data` attribute. In this case, the `lat` and `lon` attributes are not longer necessary - The Google map component will be, by default, centered on the JViews data.

You can also indicate that you allow the user to move the overlaid symbols with the `nodeMovable` attribute. If this flag is set, each interaction of the user will affect the server side location of symbols, effectively modifying their location.

Using JViews map layers and Google Maps controls

```
<jvmf:googleView id="gmapID"
      key="some Google Maps Key"
      style="width:500px;height:500px" data="/data/usa.idpr"
      controls="GLargeMapControl,GOverviewMapControl" nodeMovable="true"
      level="7" />
```

The only other JViews component that can be used in relation with the Google Maps view is the legend tool. This allows the user to select which overlaid layers should be visible. All the other JViews interactors and dependent views will have no effect.

Using JViews map layers tool with a Google Maps view

```
<jvmf:layerTool id="layerTool" title="Google View Layers" viewId="gmapID"
        enabled="true" />
```

Controlling the display of the view

The `boundingBox` property specifies the area in manager coordinates represented by the view. This property can be used to set the initial visible area.

The `changeBoundingBox` property can be used during a JSF action to reset or modify the visible area.

JViews Maps

This section shows the use of the `boundingBox` and `changeBoundingBox` properties.

Setting the initial visible area of the view

The following code example shows how to control the initial display of the view.

```
<jvmf:mapView [...] boundingBox="0,0,100,200"/>
```

Resetting or changing the visible area of the view

The following code example shows how to reset or change the visible area of the view.

```
public class MapBean {
  [...]

  public void changeBoundingBox() {
    IlvFacesMapsView jsfView = getJSFViewComponent();
    jsfView.setBoundingBox(new IlvRect(0,0,100,100));
  }
}
```

JViews Diagrammer

This section shows the use of the `boundingBox` and `changeBoundingBox` properties.

Setting the initial visible area of the view

The following code example shows how to control the initial display of the view.

```
<jvdf:diagrammerView [...] boundingBox="0,0,100,200"/>
```

Resetting or changing the visible area of the view

The following code example shows how to reset or change the visible area of the view.

```
public class DiagrammerBean {
  [...]

  public void changeBoundingBox() {
    IlvFacesDHTMLDiagrammerView jsfView = getJSFViewComponent();
    jsfView.setBoundingBox(new IlvRect(0,0,100,100));
```

```
  }
}
```

At the JViews Framework level

This section shows the use of the `boundingBox` and `changeBoundingBox` properties.

Setting the initial visible area of the view

The following code example shows how to control the initial display of the view.

```
<jvf:view [...] boundingBox="0,0,100,200"/>
```

Resetting or changing the visible area of the view

The following code example shows how to reset or change the visible area of the view.

```
public class FrameworkBean {
  [...]

  public void changeBoundingBox() {
    IlvFacesDHTMLView jsfView = getJSFViewComponent();
    jsfView.setBoundingBox(new IlvRect(0,0,100,100));
  }
}
```

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`.

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.

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

To specify fixed zoom levels, use the `zoomLevels` property.

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.

JViews Diagrammer

This section shows how to use these properties in JViews Diagrammer Faces.

Removing the constraint on the manager content

```
<jvdf:diagrammerView constrainedOnContents="false" [...] />
```

Customizing the minimum and maximum zoom levels in free zoom mode

```
<jvdf:diagrammerView minZoomLevel="2" maxZoomLevel="20" [...] />
```

Specifying fixed zoom levels

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

At the JViews Framework level

This section shows how to use these properties at the JViews Framework level.

Removing the constraint on the manager content

```
<jvf:view constrainedOnContents="false" [...] />
```

Customizing the minimum and maximum zoom levels in free zoom mode

```
<jvf:view minZoomLevel="2" maxZoomLevel="20" [...] />
```

Specifying fixed zoom levels

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

Zoom levels and dynamic layers

Fixed zoom levels, which should be used if the view is tiled, must be set so that the client can cache the tiles. This can be achieved in two different ways:

- Specify the thin client property scales in the Map Builder.
- Specify the zoom levels in the JSP™ tag using the `zoomLevels` property, for example:

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

Dynamic layers can also be specified in the Map Builder by setting the THIN_CLIENT_BACKGROUND property to `false`. Layers that have their THIN_CLIENT_BACKGROUND property set to `true` are then taken as static layers.

Note: The JSP tag attribute overrides the scales and layer properties set in the Map Builder. Be careful when setting the `zoomLevels` property and the `staticLayers` property, if your map already contains scales and static layers specified in the Map Builder.

Tiling the view

To implement tiling in the view you must specify the tile size. Other attributes, that is, the zoom levels and the dynamic or static layers, must be set to make tiling fully operational.

1. ## Specifying the tile size

   To make tiling available in the view, you must specify a tile size. The tile size must be carefully chosen because it can have a considerable and potentially critical impact on performance. The larger the number of tiles needed because of their size relative to the size of the view to be covered, the more simultaneous requests to be addressed to the image servlet. There will also be more graphic objects to manage on the client side.

   If a server-side caching mechanism is implemented, such as pregenerated tiles, the size must be consistent with the configuration of the server-side caching mechanism. See `IlvTileManager` for more details about server-side caching mechanisms.

   Setting the tile size parameter in JViews Maps

   ```
   <jvmf:mapView [...] tileSize="256"/>
   ```

   Setting the tile size parameter in JViews Diagrammer

   ```
   <jvdf:diagrammerView [...] tileSize="256"/>
   ```

   Setting the tile size parameter at the JViews Framework level

   ```
   <jvf:view [...] tileSize="256"/>
   ```

2. ## Specifying the zoom levels

   Specify the `zoomLevels` attribute to allow the client to cache the tiles for predefined levels.

   See *Zoom constraints*.

3. ## Specify the dynamic and static layers

   Specify which layers are subject to changes (dynamic layers) and which layers are not supposed to change (static layers). Static layers can be tiled and cached.

   The `staticLayersCount` attribute allows you to specify how many layers at the bottom of the JViews Faces view component are static.

   Specifying static layers in JViews Maps

   The static and dynamic layer list is determined automatically by `IlvFacesMapsServletSupport` according to the value of the layer style property `ThinClientBackground`.

   Specifying static layers in JViews Diagrammer

   ```
   <jvdf:diagrammerView [...] staticLayersCount="3"/>
   ```

Specifying static layers at the JViews Framework level

```
<jvf:view [...] staticLayersCount="3"/>
```

For more information on the use of tiling for building Web applications, see Tiling.

Visible layers

The `visibleLayers` property contains the list of the names of the visible manager layers displayed by the view.

At first, all the layers are visible by default. To specify the visible layers at initialization, use the `visibleLayers` JSP™ tag attribute.

Then, if a JSF action adds or removes some layers, the values specified by this property can be updated.

Visible layers in the map view
To specify the initial visible layers:

```
<jvmf:mapView [...] visibleLayers="layer1,layer2,layer3" />
```

To update the list of visible layers:

```
public class MapBean {
  [...]

  public void addLayer() {
    IlvFacesMapsView jsfView = getJSFViewComponent();
    //Adds a new visible layer.
    ArrayList list = jsfView.getVisibleLayers();
    list.add(newLayer.getName());
    jsfView.setVisibleLayers(list);
  }
}
```

Visible layers in the diagram view
To specify the initial visible layers:

```
<jvdf:diagrammerView [...] visibleLayers="layer1,layer2,layer3" />
```

To update the list of visible layers:

```
public class DiagrammerBean {
  [...]

  public void addLayer() {
    IlvFacesDHTMLDiagrammerView jsfView = getJSFViewComponent();
    //Adds a new visible layer.
    ArrayList list = jsfView.getVisibleLayers();
    list.add(newLayer.getName());
    jsfView.setVisibleLayers(list);
  }
}
```

Visible layers at the JViews Framework level
To specify the initial visible layers:

```
<jvf:view [...] visibleLayers="layer1,layer2,layer3" />
```

To update the list of visible layers:

```
public class FrameworkBean {
  [...]

  public void addLayer() {
    IlvFacesDHTMLView jsfView = getJSFViewComponent();
    //Adds a new visible layer.
    ArrayList list = jsfView.getVisibleLayers();
    list.add(newLayer.getName());
    jsfView.setVisibleLayers(list);
  }
}
```

Image maps

The image map allows you to have images on the client-side with an attached map that points out certain hot spots or clickable areas. A typical use case for image maps is for displaying tooltips.

The role of the image map generator is to configure the attributes and JavaScript™ handlers for each zone of the image map.

See `IlvSDMImageMapAreaGenerator` and the associated sample **Diagram Gallery** for details of how to implement an image map object in JViews Diagrammer.

See `IlvImageMapAreaGenerator` and the associated sample **Using a Manager View** for details of how to implement an image map object at the JViews Framework level.

JViews Diagrammer

Adding and displaying an image map

To add an image map and to display it, use the following code.

```
<jvdf:diagrammerView [..] generateImageMap="true"
imageMapGenerator="#{diagrammerBean.imapGenerator}"
imageMapVisible="true"/>
```

Showing or hiding an image map

You can use the JavaScript representation of the view to show or hide the image map.

```
<jvdf:diagrammerView [..] id="view"/>
<jv:imageButton id="bImgMap"

[...]
                           onclick="view.setImageMapVisible(bImgMap.isSelected
())"
                      toggle="true"
                      message="Show/Hide Tooltips" />
```

At the JViews Framework level

Adding and displaying an image map

To add an image map and to display it, use the following code.

```
<jvf:view [..] generateImageMap="true"
imageMapGenerator="#{frameworkBean.imapGenerator}"
imageMapVisible="true"/>
```

Showing or hiding an image map

You can use the JavaScript representation of the view to show or hide the image map.

```
<jvf:view [..] id="view" />
<jv:imageButton id="bImgMap"

[...]
```

```
                            onclick="view.setImageMapVisible(bImgMap.isSelected
())"
                        toggle="true"
                        message="Show/Hide Tooltips" />
```

Hiding an image map for use with interactors

The image map must be hidden to use interactors. The following code sample shows how to hide the image map when another button in the same button group is clicked.

```
<jv:imageButton id="bZoom"

[...]
                            onclick="view.setInteractor(zoomInteractor)"
                        buttonGroupId="interactors"
                        message="Zoom" />
<jv:imageButton id="bImgMap"

[...]
                            onclick="view.setImageMapVisible(bImgMap.isSelected
())"
                        buttonGroupId="interactors"
                        doActionOnBGDeselect="true"
                        message="Show/Hide Tooltips" />
```

Note: When the image map is displayed, the current interactor is disabled. To use interactors, the image map must be hidden.

See JavaScript objects for more details on the client-side representation of JSF-compatible components.

Adding a popup menu

The popup menu component allows you to display a static or contextual popup menu when the application user right-clicks in the view.

For use of menus in Facelets environments, see also *Supporting Facelets and Trinidad*.

Popup menu tag in the view tag

Since the popup menu is attached to a view, its JSP™ tag must be enclosed in the JSP tag of the view.

The popup menu can be contextual or static. The following examples show contextual popup menu tags used in the view tag.

The following code is for JViews Diagrammer.

```
<jvdf:diagrammerView [...] >
   <jvf:contextualMenu [...]/>
</jvdf:diagrammerView>
```

The following code is for JViews Framework.

```
<jvf:view [...] >
  <jvf:contextualMenu [...]>
</jvf:view>
```

Static popup menu

The menu displayed by the popup menu is static and fully on the client side.

To define a menu and menu items in JViews Diagrammer use the `menu`, `menuItem`, and `menuSeparator` tags as shown in the following example.

```
      <jvf:contextualMenu
        <jv:menu label="root">
          <jv:menuItem label="Zoom ..."
                       onclick="zoomButton.doClick()"
                       image="images/zoomrect.gif" />
          <jv:menuItem label="Pan ..."
                       onclick="panButton.doClick()"
                       image="images/pan.gif"/>
          <jv:menuSeparator/>
          <jv:menuItem label="Zoom In"
                       onclick="viewID.zoomIn()"
                       image="images/zoom.gif" />
          <jv:menuItem label="Zoom Out"
                       onclick="viewID.zoomOut()"
                       image="images/unzoom.gif"/>
          <jv:menuItem label="Zoom to Fit"
                       onclick="viewID.showAll()"
```

```
                        image="images/zoomfit.gif"/>
          <jv:menuSeparator/>
          <jv:menuItem label="Select"
                        actionListener="#{diagrammerBean.action}"
                        actionListener="#{ganttBean.action}"
                        image="images/arrow.gif"
                        invocationContext="IMAGE_SERVLET_CONTEXT" />
        </jv:menu>
      </jvf:contextualMenu>
```

To define a menu and menu items at the JViews Framework level use the `menu`, `menuItem`, and `menuSeparator` tags as in the following example.

```
<jvf:contextualMenu
        <jv:menu label="root">
          <jv:menuItem label="Zoom ..."
                        onclick="zoomButton.doClick()"
                        image="images/zoomrect.gif" />
          <jv:menuItem label="Pan ..."
                        onclick="panButton.doClick()"
                        image="images/pan.gif"/>
          <jv:menuSeparator/>
          <jv:menuItem label="Zoom In"
                        onclick="viewID.zoomIn()"
                        image="images/zoom.gif" />
          <jv:menuItem label="Zoom Out"
                        onclick="viewID.zoomOut()"
                        image="images/unzoom.gif"/>
          <jv:menuItem label="Zoom to Fit"
                        onclick="viewID.showAll()"
                        image="images/zoomfit.gif"/>
          <jv:menuSeparator/>
          <jv:menuItem label="Select"
                        actionListener="#{frameworkBean.action}"
                        actionListener="#{ganttBean.action}"
                        image="images/arrow.gif"
                        invocationContext="IMAGE_SERVLET_CONTEXT" />
        </jv:menu>
      </jvf:contextualMenu>
```

Contextual popup menu

The popup menu is dynamically generated on the server side by a menu factory depending on:

- The `menuModelId` property of the current interactor set on the view.
- The object selected when the application user triggers the popup menu.

JViews Diagrammer

To specify the factory use the `factory` or the `factoryClass` attribute of the contextual popup menu tag.

```
<jvf:contextualMenu factory="#{bean.factory}"/>
<jvf:contextualMenu factoryClass="com.xyz.demo.DemoFactory"/>
```

At the JViews Framework level

To specify the factory use the `factory` or the `factoryClass` attribute of the contextual popup menu tag.

```
<jvf:contextualMenu factory="#{bean.factory}"/>
<jvf:contextualMenu factoryClass="com.xyz.demo.DemoFactory"/>
```

The factory must implement the `IlvMenuFactory` interface.

Styling the popup menu

The popup menu is stylable by setting the following popup menu properties to a CSS class name:

- `ItemStyleClass`: the base CSS class name applied to a menu item.
- `itemHighlightedStyleClass`: the style applied over the base style when the cursor is over the item.
- `itemDisabledStyleClass`: the style applied over the base style when the cursor is disabled.

The following set of code examples shows CSS styling in a popup menu.

```
<html>
  [...]
<style>
.menuItem {
  background: #21bdbd;
  color: black;
  font-family: sans-serif;
  font-size: 12px;
}
.menuItemHighlighted {
  background: #057879;
  color: white;
}
.menuItemDisabled {
  background: #EEEEEE;
  font-style: italic;
  color: black;
}
</style>
  [...]
```

Then continue with the code for a specific JViews Faces component.

For JViews Diagrammer

```
  [...]
  <jvf:contextualMenu  itemStyleClass="menuItem"
                       itemHighlightedStyleClass="menuItemHighlighted"
                       itemDisabledStyleClass="menuItemDisabled" />
```

At the JViews Framework level

```
  [...]
  <jvf:contextualMenu  itemStyleClass="menuItem"
                       itemHighlightedStyleClass="menuItemHighlighted"
                       itemDisabledStyleClass="menuItemDisabled" />
```

Adding a legend

A legend displays and controls the visibility of the layers that compose the map. The legend is connected with the view in the usual way by using an identifier.

Connecting a legend to a view

```
<jvmf:mapView id="map" [...] />
<jvmf:layerTool viewId="map" [...]  enabled="true" />
```

You can use the legend tool in read only mode (to display which layers are visible) by setting its `enabled` property to *false*.

Note: For performance related reasons tiles are often cached on the client and/or server side. Therefore, if you are using a view in tiled mode, you cannot control the background layer visibility because the layer images would be inconsistent with the cached tiles.

The legend tool is based on an HTML TABLE element. You can style the various elements of the legend using CSS styling mechanisms. For example, if you declare your legend as follows:

```
<jvmf:layerTool styleClass="legendStyle" viewId="map"/>
```

Then you can create the following style rules to choose the appearance of the table header, background, cells, and so on.

```
. legendStyle {background: #d0d0d0; }
table.legendStyle {border-collapse: collapse; border: thin solid gray}
td.legendStyle {background: #d0d0d0; font-weight: normal ;}
thead.legendStyle {background: #d00000; font-weight: bold ;}
```

Adding a Message Box

To connect a message box to a view, use the following code:

```
<jvdf:diagrammerView [...]  messageBoxId="messageBox"/>
<jv:messageBox id="messageBox" [...]  />
```

The messages issued are now displayed in the message box.

Adding an Overview

An overview displays the global area and a rectangle corresponding to the area visible in the main view. You can move this rectangle to change the area visible in the main view. This overview is connected with the view in the usual way with the identifier.

```
<jvdf:diagrammerView id="diagrammer" [...] />
<jvf:overview viewId="diagrammer" [...] />
```

Adding a Pan Tool and a Zoom Tool

The `zoomTool` component that shows a set of buttons. Each button corresponds to a zoom level; clicking the button will zoom the view to this zoom level. The button corresponding to the current zoom level is visually different from others so that you can tell what the current zoom level is. The component can be vertical or horizontal.

The `panTool` component is a component that allows you to pan the view in all directions.

The connection to the view will be done by setting the identifier of the view to the `viewId` property of the tools.

```
<jvdf:diagrammerView id="diagrammer" [...] />
<jvf:panTool viewId="diagrammer" [...] />
<jvf:zoomTool viewId="diagrammer" [...] />
```

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.

The following code example shows server-side caching of tiles.

In JViews Maps

```
<jvmf:mapView tileManager="#{mapBean.tileManager}" [...] />
```

In JViews Diagrammer

```
<jvdf:diagrammerView tileManager="#{diagrammerBean.tileManager}" [...] />
```

At the JViews Framework level

```
<jvf:view tileManager="#{frameworkBean.tileManager}" [...] />
```

See Server-side caching and the tile manager for more information.

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 expires, 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:

In JViews Maps

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

In JViews Diagrammer

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

At the JViews Framework level

```
<jvf:view [...] 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 IBM® ILOG® 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.

The map view as a Diagrammer view

The map view component is a specialized subclass of the Diagrammer view component. For more information, refer to *Creating a diagram view*.

JViews Maps project

The easiest way to configure the content of a map view is to set a Map Builder IVL file or a JViews Diagrammer project to the map view component. This is done with the `map` attribute of the tag that points to a `.ivl` file or a `.idpr` file, as shown in the following example.

Building a Map Builder project

```
<jvmf:mapView id="map"
      map="data/usa.ivl"
      style="width:800;height:400" />
```

For more information about the Map Builder, see Using the Map Builder.

JViews Diagrammer Designer Project

The easiest way to configure the style and the data source of a diagram component is to set a JViews Diagrammer Designer project to the diagram view component. This is done with the `data` attribute of the tag that points to an `idpr` file.

```
<jvdf:diagrammerView id="diagrammer"
                data="data/diagrammer.idpr"
                style="width:800;height:400" />
```

You can set the CSS style sheet and data source separately.

Data Source Binding

If a project is not already set and you want to set a data source to a diagram, a data source component should be connected to the diagram.

Using an XML File

An easy way to connect to a data source is to use an XML file in diagram format.

```
<jvdf:diagrammerView id="diagrammer" data="data/molecule.xml"/>
```

Note: If your XML file is not in diagram format, you can use the XMLDataSource component to specify an XSLT file.

Using a Value Binding

Another way to specify a data source is to use a value binding. In this case, the data model will be provided by a Bean property:

```
<jvdf:diagrammerView [...] data="#{diagrammerBean.dataSource}" />
```

The Bean should then provide the data model through its `getDataSource` method:

```
public IlvDiagrammerDataSource getDataSource() {
  if (dataSource == null)
      dataSource = createDataSource();
  return dataSource;
}
```

Note: The JViews Diagrammer Faces component properties are all bindable.

To use the value binding attribute, the Bean must be declared in the `faces-config.xml` file or the `managed-beans.xml`:

```
<faces-config>
  <managed-bean>
    <description>A diagram component demo bean</description>
```

```
    <managed-bean-name>diagrammerBean</managed-bean-name>
    <managed-bean-class>diagrammerBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

For further information about these configuration files, see the *http://java.sun.com/j2ee/javaserverfaces/reference/index.html*JavaServer Faces specifications.

DataSource and XMLDataSource Components

Another way of setting a data source to a diagram view is to use the `dataSource` and `XMLDataSource` components. These components allow you to create and configure a data source. The data source is stored in memory and is ready to be set on a diagram component.

Setting a data source on a diagram component

```
<jvdf:XMLDataSource filename="data/molecule.xml" id="xmlDataSource" />

<jvdf:dataSource value="#{diagrammerBean.datasource}" />

<jvdf:diagrammerView id="diagrammer" data="xmlDataSource" [...] />

<h:commandButton type="button" value="Set XML Data Source"
 onclick="diagrammer.setDataSourceId('xmlDataSource')" />

<h:commandButton type="button" value="Set Bound Data Source"
 onclick="diagrammer.setDataSourceId('dataSource')" />
```

This example creates two data sources: one filled from an XML file and another one from a bound diagram data source.

The two data sources are present in memory. It is then possible to query the server for switching the data source and updating the image without a complete page refresh by clicking one of the command buttons. To perform this task, use the client-side JavaScript proxy of the diagram view.

The initial data source of the diagram view is configured through the `data` tag attribute that must match the `id` attribute of the desired data source component.

To learn how to use these proxies, see JavaScript objects.

The `diagrammer` property allows you to bind an existing `IlvDiagrammer` instance to be reused by the `diagrammerView` component.

```
<jvdf:diagrammerView [...] diagrammer="#{diagrammerBean.diagrammer}" />
```

Styling with CSS

After you set the data source, you can customize the way it is displayed. You can use Cascading Style Sheets (CSS) to style your data. CSS can be applied with a `styleSheets` attribute:

```
<jvdf:diagrammerView id="diagrammer" [...] styleSheets="data/diagrammer.css"
/>
```

The CSS file must be present in the data directory of the Web application. The style sheet file specification can also be a value binding, that is, a value provided by a Bean.

Installing Interactors

You can now install interactors on the view to interact with it:

```
<jvf:zoomInteractor id="zoom" />
<jvdf:diagrammerView interactorId="zoom" [...] />
```

The link between the view and its interactor is done through the identifier of the interactor. It is now possible to zoom on a view area by dragging a rectangle on the view.

Select Interactor

The select interactor allows you to select one or more objects and move them without performing a full refresh of the page.

To define one object, use the following tag:

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

To set the object on the view, use the following tag:

```
<jvdf:diagrammerView id="thediagrammer" interactorId="select"/>
```

Note: If you just want to trigger a server-side action when an object is clicked, use the `nodeOrLinkSelectInteractor` instead.

Move Selection

The select interactor allows you to move the selection if the following conditions are met:

- The `moveAllowed` property of the interactor is set to `true` (default value).
- The server-side object is movable.
- The node layout is disabled.

Basic Selection Management Configuration

You can customize the way the selection is performed and displayed by using a facet on the `diagrammerView` tag, as follows:

```
<jvdf:diagrammerView id="thediagrammer" interactorId="select">
  <f:facet name="selectionManager">
      <jvdf:selectionManager imageMode="false" […] />
  </f:facet>
</jvdf:diagramerView>
```

The selection manager has two display modes:

- image mode

  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 mode

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.

The default mode is the image mode.

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

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

Information on Selection

You can register a listener that will be called when the selection changes:

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

The `onSelectionChanged` attribute value is JavaScript™ code that is called when the selection has changed. The execution context defines the variable selection, which stores the current selection as an array of `IlvSelectionRectangle` instances.

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());
}
```

Besides ID and bounds properties of the selected object, you might want to get also the properties of the selected node or link in the JViews Diagrammer model.

This can be done by configuring a property accessor on the selection manager:

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

With:

```
public class ServerBean {
  private IlvFacesDiagrammerPropertyAccessor accessor =
  new IlvFacesDiagrammerPropertyAccessor();
  public IlvFacesDiagrammerPropertyAccessor getPropertyAccessor() {
    return accessor;
  }
}
```

The `IlvFacesDiagrammerPropertyAccessor` contains several methods that can be either called or redefined to configure or specialize the way it gives access to model properties.

Once done, in the JavaScript you can access all the methods of the `IlvSelectionRectangle` and do the following:

```
// 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]));
  }
}
```

In addition, if the `diagrammerView` has been set as editable:

```
<jvdf:diagrammerView editable="true" [...] />
```

you can also set properties on the client such that they can be committed back to the model on the server. You can do this by using the following code:

```
// Modify a property on the first selected object
thediagrammer.getSelectionManager().getSelection()[0].
setObjectProperty("propertyName","propertyValue");
// [other modifications]
thediagrammer.getSelectionManager().
     commitSelectionProperties(true, oncompleted, onfailed);
```

where:

- `oncompleted` is a JavaScript function that is called when the server has completed the changes, to handle errors that may have occurred while setting the new values. The parameter of the `oncompleted` method is an array of `IlvSelectionPropertiesError` objects.
- `onfailed` is a JavaScript function that is called when the commit could not occur due to network problems.

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.

Select an Object by Its Identifier

Objects can be selected on the client side by their identifier by means of the following JavaScript method: `IlvAbstractSelectionManager.selectById(id, extend)`.

The identifier of an object is retrieved through the SDM model (see Implementing the behavior of data model objects). You can also retrieve the identifier by using directly the JViews Diagrammer method `getID(java.lang.Object)`.

You can select one object by means of the following JavaScript method call:

```
thediagrammer.getSelectionManager().selectById("nodeId");
```

This method call deselects the objects currently selected and selects the object with the identifier `nodeId`. You can extend or reduce the selection by selecting or deselecting a node as follows:

```
thediagrammer.getSelectionManager().selectById("nodeId", true);
```

This method call keeps the existing selection and selects the object with the identifier `nodeId` if it is not already selected, otherwise it will deselect it.

Clear the Selection

To clear the selection use the following JavaScript method call:
`IlvAbstractSelectionManager.deselectAll()`

For example:

```
thediagrammer.getSelectionManager().deselectAll();
```

Select all the Objects

To select all the selectable objects use the following JavaScript method:
`IlvAbstractSelectionManager.selectAll()`

For example:

```
thediagrammer.getSelectionManager().selectAll();
```

Note: If the select interactor is set on the view and if the view has the focus, you can use CTRL+A to select all the objects.

Image Mode or Rectangle 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.

Rectangle mode offers basic graphic feedback but better performance.

Move Selection

The select interactor also allows you to move the selection if:

- The `moveAllowed` property of the interactor is set to `true` (default value).

- The server-side object is movable.
- The node layout is disabled.

Creating nodes and links

If you want to create nodes and/or links on the view, you have to set the editable mode on the view component as follows:

```
<jvdf:diagramerView id="thediagrammer" editable="true" [...] />
```

For example, by calling the following extract of JavaScript™ in response to a user action, an interactor will be set on the view. This interactor allows you to create a link with the `thetag` tag in the JViews Diagrammer model attached to the view.

```
thediagrammer.setCreateLinkInteractor("thetag", true);
```

When the second parameter is set to `true`, the `create` action is available only once. If you want to have the interactor available permanently, you can ignore that parameter.

Similarly, you can set a node creation interactor:

```
thediagrammer.setCreateNodeInteractor("anothertag", true);
```

In addition to the tag name, the interactor can be configured to set some initial properties on the selected object at creation time:

```
var properties = {propertyName1: "propertyValue1", propertyName2:
"propertyValue2"};
thediagrammer.setCreateNodeInteractor("anothertag", true, properties);
```

In order to work correctly, a property accessor must be set and configured on the facet (see Information on selection).

Deleting selected nodes and links

If you want to be able to delete selected nodes and links on the view, you have to set the editable mode on the view component as follows:

```
<jvdf:diagrammerView id="thediagrammer" editable="true" [...] />
```

This allows you to ask for deletion of the selected objects on the client side by using the following JavaScript™ code:

```
thediagrammer.getSelectionManager().deleteSelection();
```

Dashboard diagram

The `jvdf:dashboardView` component allows you to display a dashboard diagram. This JSF component manages an `IlvDashboardDiagram` instance to display the image. This class is an `IlvDiagrammer` subclass, therefore the `diagrammerView` and the `dashboardView` share a lot of properties. This component can be used in the same way as the `diagrammerView`, except that the only way to set data is to use the `data` attribute (no project, style sheets properties). This attribute only accepts `idbd` files.

```
<jvdf:dashboardView [...] data="/data/dashboard.idbd" />
```

The `dashboardDiagram` property allows you to bind an existing `IlvDashboardDiagram` instance to be reused by the `dashboardView` component.

```
<jvdf:dashboardView [...] dashboardDiagram="#{dashboardBean.dashboard}" />
```

Note: Palettes files (JARs) must be in the application classpath (usually in the `WEB-INF/lib` of the WAR file) for the `dashboardView` component to load the dashboard.

JavaScript objects

Each time a JViews Maps Faces component is created, a corresponding JavaScript™ object is also created. You can access this object through a global JavaScript variable whose name is the same as the `id` attribute of the tag. For example, the tag:

```
<jvmf:mapView id="map" [...] />
```

will be rendered as the following JavaScript code:

```
map = new IlvDiagrammerViewProxy ('map', ...);
map.setServletClass("ilog.views.maps.servlet.IlvFacesMapsServlet");
```

See the documentation of the Java™ API (`overview-summary`) of each renderer to know which JavaScript proxy will be generated for this component.

You can modify the object locally by using a set of methods attached to this object. For further information about available JavaScript objects, see the DHTML reference documentation of JViews Maps.

The following example defines a button that dynamically installs a zoom interactor on the view without a server round trip.

Defining a dynamic zoom interactor button

```
<jvf:panInteractor id="pan"
<jvf:imageButton [...] onclick="map.setInteractor(pan)"  />
<jvmf:mapView id="map" [...] />
```

At rendering time, an `IlvDiagrammerViewProxy` JavaScript object is created, accessible through the JavaScript variable.

Then, since a JavaScript object named `pan` has been created in the same way, you can directly set this interactor with the `setInteractor` method.

Additionally, the behavior of these JavaScript objects is to keep their state, so that if a submit request is issued, the state of the object is sent to the server. This behavior makes sure that the client and the server remain coherent.

For further information about available JavaScript objects, see the JavaScript API DHTML reference documentation of JViews Maps.

Contexts for actions on the view

In this section

Overview

JavaServer Faces lifecycle context
Explains how to install a select object interactor and a listener in the JSF context.

Image servlet context
Describes the value change listener and interactor in the image servlet context.

Overview

Actions executed in response to interactions on the view can be executed in two different contexts: JavaServer™ Faces lifecycle or image servlet. The execution context can be configured by setting the `invocationContext` attribute on the JSF interactor components.

The value change listeners registered in the interactor can determine whether they are called in a JSF context or in an image servlet context with the following code:

Determining in Which Context a Value Change Listener is Called

```
IlvObjectSelectInteractor source =
        (IlvObjectSelectInteractor)valueChangeEvent.getSource();
boolean jsfContext = source.getInvocationContext() ==
        IlvDHTMLConstants.JSF_CONTEXT;
```

This section shows the differences between the two invocation contexts through the execution of an action when a node is selected.

JavaServer Faces lifecycle context

This topic shows you the JViews Faces code for installing a select object interactor and a listener. It also shows you the Java™ code for writing a value-change event listener.

In JViews Diagrammer

To select an SDM node in a view, a select object interactor must be installed on the diagram component view. The `value` property of the interactor holds the `IlvSDMNode` object that was clicked. Thus, a `valueChangeListener` can be registered to handle the selection event.

Installing a select node or link interactor and a listener

```
<jvdf:nodeOrLinkSelectInteractor id="objSelect"
     valueChangeListener="#{diagrammerBean.onSelectNode}"
                 invocationContext="JSF_CONTEXT"/>

<jvdf:diagrammerView id="diagrammer" interactorId="objSelect" [...] />
```

Note: `JSF_CONTEXT` is the default value, so the `invocationContext` attribute could have been omitted.

Java code of the value-change event

The Java code of the value change event listener is:

```
  public void onSelectNode(ValueChangeEvent event) {
    IlvSDMNode node = (IlvSDMNode) event.getNewValue();
    if (node != null) {

         //The source of the event is the interactor
         IlvFacesNodeOrLinkSelectInteractor source =
              (IlvFacesNodeOrLinkSelectInteractor)valueChangeEvent.getSource
();

               //Retrieve the JSF view connected to the interactor
               IlvFacesDiagrammerView jsfDiagrammer =
                 (IlvFacesDiagrammerView)source.getView();

      try {
        //Retrieve the IlvDiagrammer wrapped by the JSF component.
        IlvDiagrammer diagrammer = jsfDiagrammer.getDiagrammer();

                //Select the clicked object
        diagrammer.deselectAll();
        diagrammer.setSelected(node, true);

      } catch (Exception e) {
        e.printStackTrace();
```

```
      }
    }
  }
```

At the JViews Framework level

To select a graphic object in a view at the JViews Framework level, a select object interactor must be installed on the view. The `value` property of the interactor holds the `IlvGraphic` object that was clicked. Thus, a `valueChangeListener` can be registered to handle the selection event.

Installing a select object interactor and a listener

```
<jvf:objectSelectInteractor id="objSelect"
     valueChangeListener="#{frameworkBean.selectObject}"
                 invocationContext="JSF_CONTEXT"/>

<jvf:view id="view" interactorId="objSelect" [...] />
```

Note: `JSF_CONTEXT` is the default value, so the `invocationContext` attribute could have been omitted.

Java code of value-change event

The Java code of the value change event listener is:

```
public void selectObject(ValueChangeEvent event) {
  Object value = event.getNewValue();
  if (value != null && value instanceof IlvGraphic) {

           //The source of the event is the interactor
           IlvFacesObjectSelectInteractor source =
                (IlvFacesObjectSelectInteractor)valueChangeEvent.getSource();


               //Retrieve the JSF view connected to the interactor
    IlvFacesView jsfView = (IlvFacesView)source.getView();

         //Retrieve the IlvManagerView wrapped by the JSF component.
    IlvManagerView managerView = jsfView.getView();

               //Select the clicked object
    IlvGraphic g = (IlvGraphic) value;
    managerView.getManager().deSelectAll(false);
    managerView.getManager().setSelected(g, true, false);
  }
}
```

Note the following concerning the use of this approach:

- Since the method is called during the JavaServer™ Faces lifecycle, there can be interaction with other JSF components.
- The form is submitted, so the complete page is reloaded.

Image servlet context

The image servlet uses the same value change listener as the JavaServer™ Faces lifecycle; there is a slight difference in the interactor, which is shown in bold in the example.

Value change listener and interactor in image servlet context (JViews Diagrammer)

```
<jvdf:nodeOrLinkSelectInteractor id="objSelect"
     valueChangeListener="#{diagrammerBean.onSelectNode}"
                 invocationContext="IMAGE_SERVLET_CONTEXT"/>

<jvdf:diagrammerView id="diagrammer" interactorId="objSelect" [...] />
```

Value change listener and interactor in image servlet context (JViews Framework level)

```
<jvf:objectSelectInteractor id="objSelect"
     valueChangeListener="#{frameworkBean.selectObject}"
                 invocationContext="IMAGE_SERVLET_CONTEXT"/>

<jvf:view id="view" interactorId="objSelect" [...] />
```

In this mode the interactor queries an image update. The server fires the value change event just before image generation.

This approach in JViews Diagrammer:

- Avoids submitting the page and refreshes the image only.
- Is outside the JSF lifecycle, so no interaction with JSF components is possible beyond the ability to retrieve the `IlvDiagrammer` object as shown in *Java code of the value-change event*..

This method at JViews Framework level:

- Avoids submitting the page and refreshes the image only.
- Is outside the JSF lifecycle, so no interaction with JSF components is possible beyond the ability to retrieve the `IlvManagerView` as shown in *Java code of value-change event*.

Integrating JViews Faces in your environment

Provides information about configuring a JSF application in the application server, session persistence, JSR 168 portlets, ICEfaces, and Facelets and Trinidad.

In this section

JViews Faces configuration at JViews Framework level
Provides required and optional settings for JViews Faces configuration at the JViews Framework level.

Session persistence
Explains how to disable session persistence.

Running JViews Faces components in JSR 168 portlets
Explains the JSR 168 requirements for JViews Faces components in portlets.

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

Supporting Facelets and Trinidad
Describes the mandatory actions required to make JViews Faces components compatible with Facelets and Trinidad, plus optional actions to specify menus.

Web Application Server support
Describes the Web Application Servers supported for deploying JViews Web applications.

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 as follows.

```
<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 if 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 follows.

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

Note: Although optional, it is recommended to set this setting always to `true`.

Session persistence

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

Often, the JViews Faces components are not serializable as they pertain to view-related abstractions which typically cannot be persistent 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.

To disable session persistence in TOMCAT at web application level:

1. Create a file `context.xml` and place it in the META-INF directory of your `.war` file.
2. Use a TOMCAT configuration setting 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 and later.

To disable session persistence in TOMCAT at web server level:

- Modify the `TOMCAT/conf/context.xml` to use this as the Session Manager definition.

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

Note: These settings apply to TOMCAT 6.0 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, see the server's 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 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 Faces components compatible with portlets.

Note: JViews 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 Faces components must be prefixed by the portlet namespace.

Scripts prefixed by a namespace

Since JViews 8.1, the servlet filter `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.

In IBM® ILOG® JViews a JavaScript action is built on a managed bean by using the static method `encodeJavaScriptVariables` of `ilog.views.faces.IlvFacesUtil`.

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

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

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

The result of calling this method is the final JavaScript action with namespace-encoded variables.

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

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

In JViews Diagrammer

Using JavaScript actions in a JSP page

```
[...]
<jvf:zoomInteractor id="zoom" />
<jv:imageButton onclick="#{diagrammerBean.setZoomAction}"/>
<jvdf:diagrammerView id="diagrammer" />
[...]
```

Using JavaScript actions in a managed bean

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

At the JViews Framework level

Using JavaScript actions in a JSP page

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

Using JavaScript actions in a managed bean

```
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:

In JViews Diagrammer

```
<jvdf diagrammerView [...] servlet=
    "ilog.views.diagrammer.faces.dhtml.servlet.IlvFacesDiagrammerServlet />"
```

At the JViews Framework level

```
<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.

Guide to using JViews components with ICEfaces

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

In this section

Settings for using JViews components in ICEfaces
Describes the settings you need to use JViews JSF components with ICEfaces.

Interoperability between JViews components and ICEfaces components
Describes the interoperability between JViews components and ICEfaces components.

Push updates to JViews components
Describes the techniques for push updates (server-initiated rendering) with JViews components.

ICEfaces software in JViews
Describes the ICEfaces binary files provided with JViews and lists the known issues.

Settings for using JViews components in 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 *JViews Faces configuration at JViews Framework level*.

For more settings and concrete examples, look at the code sample installed in ***<install-dir>*** **/jviews-maps8.6/codefragments/jsf-maps-ice**.

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 managed beans. This allows you to exchange parameters and data between the managed beans of JViews components and ICEfaces components.

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 sample ***<install-dir>*** **`/jviews-diagrammer8.6/codefragments/jsf-diagrammer-ice`** in `diagrammer.jsp`. 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 managed 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

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 Diagrammer at ***<install-dir>*** **`/jviews-diagrammer8.6/codefragments/jsf-diagrammer-ice`**.

ICEfaces software in JViews

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 `<framework-install-dir>/lib/external`. However, the full ICEfaces distribution is not included.

To get a complete or more updated distribution, you can get ICEfaces source code at *http://www.icefaces.org*.

Known ICEfaces issues

Issues may exist 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 sample installed in ***<install-dir>*** **`/jviews-diagrammer8.6/codefragments/jsf-diagrammer-ice`**.

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.

  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.

Code examples

For complete JViews Maps application examples configured for use with Facelets or Trinidad, see ***<install-dir>*** **`/jviews-maps8.6/codefragments/jsf-maps-facelets/webpages/index.xhtml`**.

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" />
```

At the JViews Framework level, the contextual menu element is `contextualMenu`.

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}" />
```

Web Application Server support

Apache Tomcat™ 6.0.14 is the reference Web Application Server (AS) shipped with IBM® ILOG® JViews 8.6.

Other Web AS have been tested, including JBoss® AS 4.2.3.GA, IBM® WebSphere® 7.0, and Oracle® Weblogic Server 10.3. The following sections give useful information you may need when deploying JViews Web applications to one of these servers.

JBoss Application Server 4.2.3.GA

- JBoss AS 4.2.3.GA includes a JSF implementation. 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 might 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 multipattern `<servlet-mapping>` elements in `web.xml`. You should use multiple `<servlet-mapping>` elements with separate patterns.

IBM WebSphere 7.0

- WebSphere 7.0 includes a JSF implementation. 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 might 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` 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 demos with invalid PPR responses, the problem is caused by an invalid XML response, which has been reported at *https://issues.apache.org* as JIRA issue TRINIDAD-1170.

Deploying a JViews Maps application as a DHTML-only thin client

Explains how to deploy JViews Maps Faces application as a DHTML-only thin client.

In this section

JavaServer Faces components as opposed to DHTML thin client
Recommends the use of DHTML-based JavaServer™ Faces (JSF) technology rather than DHTML-only thin-client technology.

Thin-client classes for the server side
Describes the classes provided to implement thin-client server-side applications.

Deploying an application as a DHTML-only thin client
Describes how to deploy a JViews Diagrammer application as a DHTML-only thin client.

JavaServer Faces components as opposed to DHTML thin client

When you build a DHTML-based Web application, you are recommended to base the application on JavaServer™ Faces (JSF) technology.

Build your application with the techniques described in *Using DHTML-based JSF Components to build Web applications*

JSF components in JViews Maps rely heavily on DHTML thin-client libraries, both on the server and the client, so you need to be familiar with the topics discussed here to be able to use the JSF components properly.

On the server side, the JSF components leverage the thin-client servlet to generate images and other kinds of output for the client side. On the client side, the JSF components use JavaScript™ classes of the DHTML thin client to provide Ajax features.

For a basic use of a JSF component, you probably do not need a full understanding of the DHTML thin client. Advanced use requires you to have a reasonable knowledge of it.

In rare cases, such as environments where JSF is not available, you might need to rely solely on the DHTML thin client.

Thin-client classes for the server side

Describes the classes provided to implement thin-client server-side applications.

In this section

Overview
Lists the classes used to implement thin-client server-side applications.

The IlvMapServlet class
Describes the IlvMapServlet class.

The IlvMapServletSupport class
Describes the IlvMapServletSupport class.

Overview

The following classes are provided for to help you implement thin-client server side applications:

Thin-Client Server Side Classes

Name	Description
`IlvMapServlet`	A ready-to-use servlet.
`IlvMapServletSupport`	Provides servlet functionality but is not a ready-to-use servlet.

The IlvMapServlet class

The `IlvMapServlet` class is a concrete servlet class that handles HTTP requests to build an image of a diagram.

This class accepts one parameter: the project file which refers to the XML data file and the style sheet on which the diagram is based. This parameter can be specified at initialization time (as a static configuration parameter of the servlet) or dynamically in the request URL.

`IlvMapServlet` provides support for client-side image maps. The image map can be used to attach actions to every node or link displayed in the diagram.

`IlvMapServlet` extends `IlvDiagrammerServlet`, it uses the same protocols as `IlvManagerServlet` to communicate with clients. `IlvMapServlet` is also subclass of `IlvSDMServlet` and inherits its session capabilities. Depending on the value of the `multiSession` configuration parameter, the JViews Maps servlet can work in either mono-session or multi-session mode. By default, the servlet runs in mono-session mode. See *Mono-session and multi-session modes* for more information.

Mono-session and multi-session modes

Name	Description
Mono-session	If `multiSession` is set to `false`, one SDM engine and one SDM view, including one grapher, are created. All the requests to the servlet will return an image of the same shared diagram. This option is appropriate if the users are not allowed to modify the contents of the diagram or if the changes should be visible by all users.
Multi-session	If `multiSession` is set to `true`, a new SDM engine, and its associated SDM view and grapher, is created for every different client session connected to the servlet. This mode is appropriate if users are allowed to modify the XML file or the style sheet.

The IlvMapServletSupport class

The class `IlvMapServletSupport` is a subclass of `IlvDiagrammerServletSupport`, it implements the functionality of the JViews Maps servlet but is not a servlet itself.

The purpose of `IlvMapServletSupport` is to let you do the following:

- Build multiplexing servlets that can handle requests for images.
- Handle other kinds of requests that call other parts of your application.

The class `IlvMapServlet` is a facade that forwards requests to `IlvMapServletSupport`.

Deploying an application as a DHTML-only thin client

Describes how to deploy a JViews Diagrammer application as a DHTML-only thin client.

In this section

JavaServer Faces components as opposed to DHTML thin client
Recommends the use of DHTML-based JavaServer™ Faces (JSF) technology rather than DHTML-only thin-client technology.

Thin-client library
Describes the elements of the IBM® ILOG® JViews Diagrammer thin-client library.

Creating a thin-client application
Shows how to create a thin-client application.

Thin-client classes for the server side
Defines the server-side thin-client classes.

Writing the client side of Web applications using JViews Diagrammer
Shows how to write the client side of your Web applications using JViews Framework DHTML libraries.

Managing selection
Describes how to use the classes that manage selection.

Creating nodes and links
Describes how to create nodes and links in your Web applications.

Deleting selected nodes and links
Shows how to delete selected nodes and links.

JavaServer Faces components as opposed to DHTML thin client

When you build a DHTML-based Web application, you are recommended to base the application on JavaServer™ Faces (JSF) technology.

Build your application with the techniques described in *Using DHTML-based JSF Components to build Web applications*

JSF components in JViews Maps rely heavily on DHTML thin-client libraries, both on the server and the client, so you need to be familiar with the topics discussed here to be able to use the JSF components properly.

On the server side, the JSF components leverage the thin-client servlet to generate images and other kinds of output for the client side. On the client side, the JSF components use JavaScript™ classes of the DHTML thin client to provide Ajax features.

For a basic use of a JSF component, you probably do not need a full understanding of the DHTML thin client. Advanced use requires you to have a reasonable knowledge of it.

In rare cases, such as environments where JSF is not available, you might need to rely solely on the DHTML thin client.

Thin-client library

The thin-client library provides the basic infrastructure for building Web applications that display a graph. In IBM® ILOG® JViews Diagrammer, the contents of the graph come from a diagram component.

To make it easier to build Web applications that display a diagram component, IBM® ILOG® JViews Diagrammer extends the generic thin-client support, providing classes that let you implement the server side of your Web applications with little or no coding.

If you want full details of lower-level thin-client support, see DHTML thin-client support in JViews Framework in *Advanced Features of JViews Framework*. Note in particular that the architecture for JViews Diagrammer is equivalent to what is shown in IBM® ILOG® JViews thin-client Web architecture in *Advanced Features of JViews Framework*, but with the following specifics:

- The JViews Application is a JViews Diagrammer application, containing an `IlvDiagrammer` instance.
- The JViews Servlet is a JViews Diagrammer servlet, based on a JViews Diagrammer server-side class.

Creating a thin-client application

To create a thin-client (Web) application, use the class `IlvDiagrammerServlet`. This class accepts JViews Diagrammer project files as input, so all you have to do to display a diagram in a thin-client application is to pass the project file as a parameter to the servlet.

An example of a thin-client application based on JViews Diagrammer is supplied in **<installdir>/jviews-diagrammer86/samples/diagrammer/thinclient/**.

This example uses the JViews Diagrammer servlet to display a workflow process in a Web browser. The user can choose different processes to display and different style sheets to render them. The elements of the workflow can also be selected.

Important: Only the browsers and browser versions listed in the Release notes are supported by the IBM® ILOG® JViews DHTML thin client.

Thin-client classes for the server side

Defines the server-side thin-client classes.

In this section

The IlvDiagrammerServlet class
Describes the `IlvDiagrammerServlet` class.

The IlvDiagrammerServletSupport class
Describes the `IlvDiagrammerServletSupport` class.

The IlvDiagrammerServlet class

The `IlvDiagrammerServlet` class in the package `ilog.views.diagrammer.servlet` is a concrete servlet class that handles HTTP requests to build an image of the diagram.

This class accepts one parameter: the project file which refers to the XML data file and the style sheet on which the diagram is based.

This parameter can be specified at initialization time (as a static configuration parameter of the servlet) or dynamically in the request URL.

The JViews Diagrammer servlet can work in two modes, depending on the value of the `multiSession` configuration parameter:

- Mono-session mode: If `multiSession` is false, only one SDM engine and one SDM view (and one grapher) are created. All the requests to the servlet will return an image of the same shared diagram. This option is appropriate if the users are not allowed to modify the contents of the diagram or if the changes should be visible by all users.
- Multi-session mode: If `multiSession` is true, a new SDM engine (with its associated SDM view and grapher) is created for every different client session connected to the servlet. This mode is appropriate if users are allowed to modify the XML file or the style sheet.

By default, the servlet runs in mono-session mode.

The class `IlvDiagrammerServlet` provides support for client-side image maps. The image map can be used to attach actions to every node or link displayed in the diagram.

The class `IlvDiagrammerServlet` is a subclass of `IlvManagerServlet` that is using the same protocols as `IlvDiagrammerServlet` to communicate with clients.

The IlvDiagrammerServletSupport class

The class `IlvDiagrammerServletSupport` is a subclass of `IlvManagerServletSupport`. This class implements the functionality of the JViews Diagrammer servlet but is not a servlet itself.

The purpose of `IlvDiagrammerServletSupport` is to let you build “multiplexing” servlets that can handle requests for images, but also other kinds of requests that call other parts of your application.

The class `IlvDiagrammerServlet` is a facade that forwards requests to the class `IlvDiagrammerServletSupport` .

Writing the client side of Web applications using JViews Diagrammer

To write the client side of your Web applications, you use the JViews Framework DHTML libraries. See DHTML thin-client support in JViews Framework in *Advanced Features of JViews Framework* for information on how to use the JViews Framework DHTML thin client.

To specify from the client side the XML, CSS or project files to be used by the JViews Diagrammer servlet, you can put parameters on the query URL with the given names (XML, CSS, project).

In addition to the features available in the JViews Framework DHTML library, you can add features that are specific to JViews Diagrammer DHTML and that are described in the following sections.

Managing selection

Selection in a DHTML thin-client application is handled through the `IlvSelectionManager` and `IlvSelectInteractor` classes.

The `IlvSelectionManager` allows you to select one or more objects in the view and to move the current selection. The `IlvSelectInteractor` directly uses the selection manager to perform these tasks.

Note: Selection makes sense in multisession mode only.

Client-side configuration

To use the selection on the client side, import the following scripts:

```
<script TYPE="text/javascript" src="script/IlvAbstractSelectionManager.js"></
script>
<script TYPE="text/javascript" src="script/framework/IlvSelectionManager.js"></
script>
<script TYPE="text/javascript" src="script/framework/IlvSelectInteractor.js"></
script>
```

To retrieve the selection manager on the client side, use the following code:

```
var selectionManager = view.getSelectionManager();
```

Server-side configuration

By default, the selection feature is not enabled in the image servlet support.

To make the selection feature available, you need to call `setSelectionEnabled(true)`.

Example:

```
public class DiagrammerServlet  extends IlvDiagrammerServlet  {

  protected IlvSDMServletSupport createServletSupport(ServletContext context)
 {
    return new DiagrammerServletSupport(context);
  }

}

public class DiagrammerServletSupport extends IlvDiagrammerServletSupport {
```

```
  public DiagrammerServletSupport(ServletContext context) {
    super(context);
    setSelectionEnabled(true);
  }
}
```

Image mode or Rectangle mode

The selection manager provides two modes for displaying the selection of objects:

- Image mode: after each selection on the client, an image request is issued.
- Rectangle mode: after each selection on the client, the bounding box of each selected object is queried to the server and displayed on the client.

  The color and thickness of the bounding box rectangles can be configured through the selection manager.

Properties

The selection servlet can be configured to provide additional information for each selected object. This information corresponds to additional properties that can be used on the client.

To do so, you need to subclass the selection support implementation to override `getAdditionalProperties(IlvSelectionResponse response, Object object)`, as in the following example.

The following selection support adds the properties of an SDM node:

```
//Subclass to override the getAdditionalProperties method
public class DiagrammerSelectionSupport extends IlvDiagrammerSelectionSupport
 {

  public DiagrammerSelectionSupport(IlvDiagrammerServletSupport support) {
    super(support);
  }

  protected ArrayList getAdditionalProperties(IlvSelectionResponse response,
   Object object) {
    ArrayList props = super.getAdditionalProperties(response, object);

    IlvSDMNode node = (IlvSDMNode) object;
    IlvDiagrammer diagrammer = (IlvDiagrammer)
       response.getProperty(DIAGRAMMER_KEY);
    IlvSDMModel model = diagrammer.getEngine().getModel();

    String names[] = model.getObjectPropertyNames(node);

    for (int i = 0; i < names.length; i++) {
      ArrayList l = new ArrayList();
      l.add(names[i]);
      l.add(model.getObjectProperty(node, names[i]));
      props.add(l);
```

```
    }

     return props;
  }

}

//Use the new selection selection support class in the servlet support.
public class DiagrammerServletSupport extends IlvDiagrammerServletSupport {

  [...]

  protected IlvSelectionSupport createSelectionSupport() {
    return new DiagrammerSelectionSupport(this);
  }

}
```

In image mode, you need to issue an additional request to get the selection information (the information is disabled by default). To force this additional request on each selection, use `view.getSelectionManager().setForceUpdateProperties(true)`.

In rectangle mode, the selection information is always enabled.

There are two ways to retrieve the selection information on the client side:

- Get the current selection at any time.

  ```
  view.getSelectionManager().getSelection()
  ```

- Register a listener for selection changes.

  The listener will be notified of the current selection.

The additional properties are available in the `properties` of a selection rectangle.

If the selection support example illustrated earlier in this section is used, the following listener example will fill a panel with properties of the selected object:

```
function showProperties(rList) {

  if (rList.length == 1) {
    var p = "<table>";

    for(var i=0; i<rList[0].getProperties(length); i++){
      var props = rList[0].getProperties();
      p += "<td>" + props[i][0]+ "</td>";
      p += "<td>" + props[i][1]+ "</td>";
    }
    p +="<table>";
    propPanel.setContent(p);
  }
}
```

Listeners

Listeners can be registered for the selection manager to keep track of the selection when the selection manager is in rectangle mode. To add a listener to the selection manager, use the following method:

```
view.getSelectionManager().addSelectionChangedListener(listener)
```

`listener` is a function with one parameter that corresponds to the selection: a list of rectangles with additional properties.

In image mode, you need to issue an additional request to get the selection information; listeners are not notified by default.

To force this second request on each selection, use:

```
view.getSelectionManager().setForceUpdateProperties(true)
```

Moving

If moving objects is allowed on the client and server sides, the `IlvSelectInteractor` allows you to drag and drop objects.

Important: To avoid user actions to be overridden by the automatic layout, the node layout must be disabled when using that feature.

Creating nodes and links

Describes how to create nodes and links in your Web applications.

In this section

Overview
Provides an overview of the class used to create nodes and links.

Client-side configuration
Shows how to configure nodes and links on the client side of the Web application.

Server-side configuration
Shows how to configure nodes and links on the server side of the Web application.

Overview

The ability of interactively creating nodes and links in a DHTML thin-client application is handled through the `IlvMakeObjectInteractor` class.

When the `IlvMakeObjectInteractor` class is set on the view and configured as a node creation interactor, it allows you to create nodes by clicking on their expected position on the view.

Alternatively, when the `IlvMakeObjectInteractor` class is configured as a link creation interactor, it allows you to create links by clicking the mouse on the origin node, dragging the mouse and then releasing it on top of the destination node.

Note: Normally, the creation of nodes and links is expected only in multisession mode.

Client-side configuration

To be able to interactively create nodes and links on the client side, import the following scripts:

```
<script TYPE="text/javascript" src="script/IlvAbstractSelectionManager.js">
</script>
<script TYPE="text/javascript" src="script/framework/IlvSelectionManager.js">
</script>
<script TYPE="text/javascript" src="script/framework/IlvMakeObjectInteractor.
js"></script>
```

The interactor can be instantiated, configured and set as follows:

```
var interactor = new IlvMakeObjectInteractor();
// optionally make it a link interactor
interactor.setLinkMode(true);
// mandatory, set the tag of the created object in the diagrammer model
interactor.setAdditionalParameters("tagname");
view.setInteractor(interactor);
```

In addition to the tag name, the interactor can be configured to set some initial properties on the selected object at creation time:

```
var properties = {propertyName1: "propertyValue1",
      propertyName2: "propertyValue2"};
interactor.setProperties(properties);
```

Server-side configuration

To be able to deal with the actions submitted by the `IlvMakeObjectInteractor` class from the client side, the image Servlet support must be configured by adding the action listener that can handle those actions.

Example:

```
public class DiagrammerServlet extends IlvDiagrammerServlet {
  protected IlvSDMServletSupport createServletSupport(ServletContext context)
 {
    return new DiagrammerServletSupport(context);
  }
}

public class DiagrammerServletSupport extends IlvDiagrammerServletSupport {
  public DiagrammerServletSupport(ServletContext context) {
    super(context);
    addServerActionListener(new IlvDiagrammerCreateActionListener());;
  }
}
```

Deleting selected nodes and links

Shows how to delete selected nodes and links.

In this section

Overview
Provides an overview of the class used to delete selected nodes and links.

Client-side configuration
Shows how to configure deletion of selected nodes and links on the client side of the Web application.

Server-side configuration
Shows how to configure deletion of selected nodes and links on the server side of the Web application

Overview

The ability to delete selected nodes and links in a DHTML thin-client application is handled through the `IlvSelectionManager` class.

Once set on the view and configured (see *Managing selection*) you can call the `deleteSelection()` method on the `IlvSelectionManager` class to delete objects that have been selected using the `IlvSelectInteractor`.

Note: Nodes and Links deletion makes sense in multisession mode only.

Client-side configuration

To be able to delete the selected nodes and links on the client side, import the following scripts:

```
<script TYPE="text/javascript" src="script/IlvAbstractSelectionManager.js">
</script>
<script TYPE="text/javascript" src="script/framework/IlvSelectionManager.js">
</script>
```

Once selected by the user, the objects can be deleted using the following line of code:

```
view.getSelectionManager().deleteSelection();
```

Server-side configuration

To be able to deal with the actions submitted by the `deleteSelection()` call from the client side, the image servlet support must be configured by adding the action listener that can handle them.

Example:

```
public class DiagrammerServlet extends IlvDiagrammerServlet {
  protected IlvSDMServletSupport createServletSupport(ServletContext context)
 {
    return new DiagrammerServletSupport(context);
  }
}

public class DiagrammerServletSupport extends IlvDiagrammerServletSupport {
  public DiagrammerServletSupport(ServletContext context) {
    super(context);
    addServerActionListener(new IlvDiagrammerDeleteActionListener());;
  }
}
```

Index

T

V

W

Z