IBM ILOG JViews Charts V8.6

Building Web Applications

Copyright

Copyright notice

Trademarks

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

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

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

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

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

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

Notices

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

Table of contents

Introducing the Web technologies used in JViews Charts

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 Charts 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 Charts 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 Charts 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 an 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.

Rich Web client
Introduces rich Web clients.

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 Charts helps you to develop such applications as:

- Rich Web Client
- Applets
- Java™ Web Start Applications

Rich Web client

JViews Charts provides a set of JSF components that relies on Ajax technology and SVG for rendering.

SVG is a W3C standard that allows for advanced drawings to be built by script on the client. Such local execution capabilities allow for advanced interaction locally. Thus they offload the servers and retain all other thin client advantages. The major issue with such an approach is that the execution environment needs to support SVG either through a plug-in as in Internet Explorer or natively as in FireFox.

See *Creating Rich Web Charts* for more information about this deployment strategy.

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 Charts application using this approach, see Developing with the JViews Charts SDK.

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 Charts application using this approach, see Developing with the JViews Charts SDK and the Java Web Start documentation at *http://java.sun.com/products/javawebstart/developers.html*.

Using DHTML-based JSF components to build Web applications

This section shows you how to use the components of IBM® ILOG® JViews Charts Faces to create JavaServer™ Pages (JSP™) that are compliant with JavaServer Faces (JSF).

In this section

Introduction
Provides an overview of the JViews Charts Faces Components.

The architecture of JViews Charts Faces
Presents an overview of JViews Charts Faces architecture.

The JViews Charts Faces component set
Presents some examples to illustrate how to use the JViews Charts Faces components

JavaScript objects
Explains the creation of JavaScript objects corresponding to JViews Charts Faces components.

Contexts for actions on the view
Describes the contexts in which actions can be executed in response to interactions 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.

Introduction

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

The architecture of JViews Charts Faces

Presents an overview of JViews Charts Faces architecture.

In this section

About support for JViews Charts Faces
Describes the components of JViews Charts Faces.

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

About support for JViews Charts Faces

JViews Charts Faces support is based on JavaServer Faces (JSF) technology and 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. The JViews Charts Faces component set includes:

- a chart
- an overview
- a legend
- a set of interactors
- a popup menu

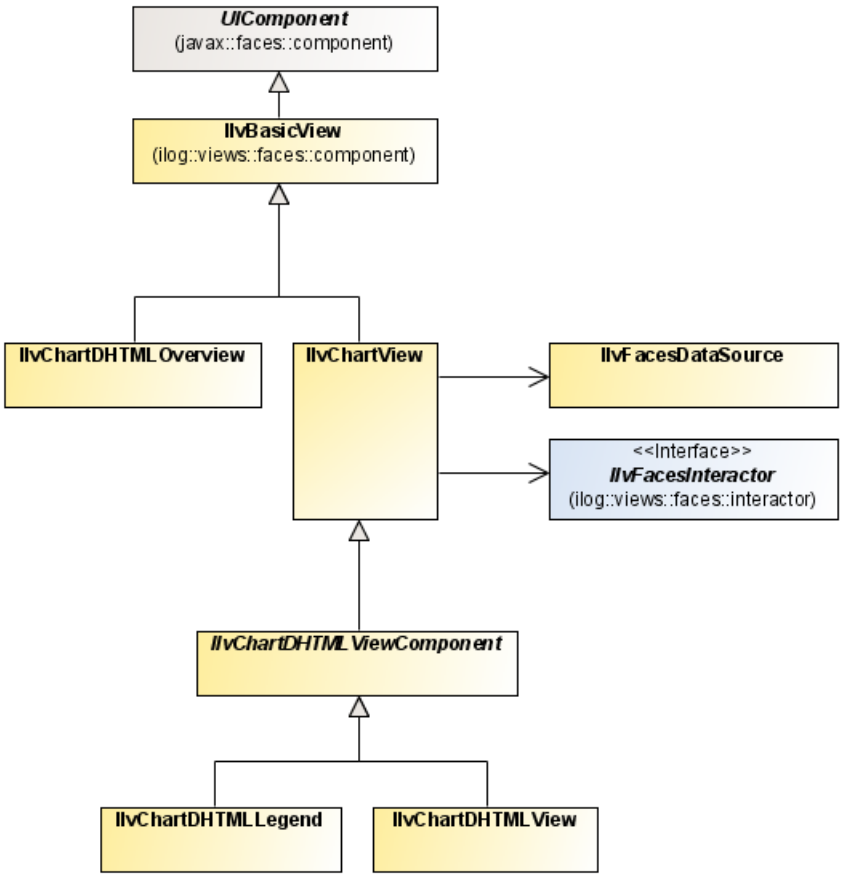
Not all the components have a visual representation. An interactor, for example, is intended only to be set on a chart view and has no visual representation.

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 issuing a server roundtrip.

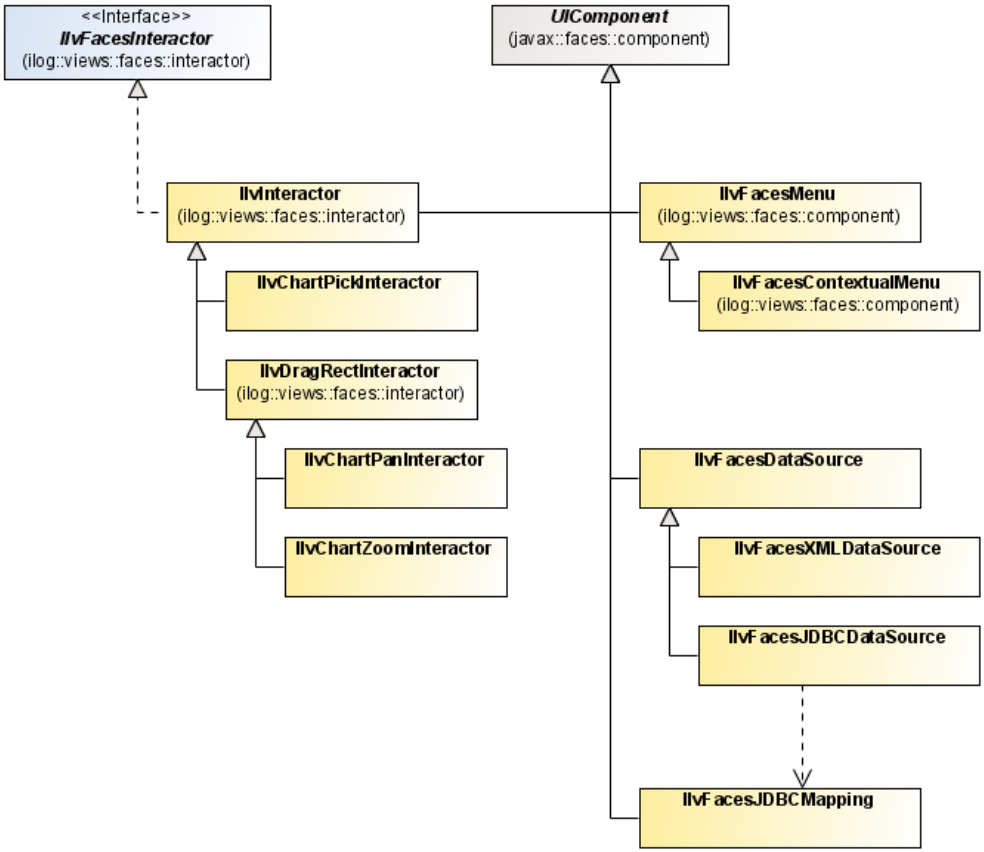
- See the Release Notes for the Web browsers and versions with which JViews Charts Faces components are compatible.

Servlet and component classes

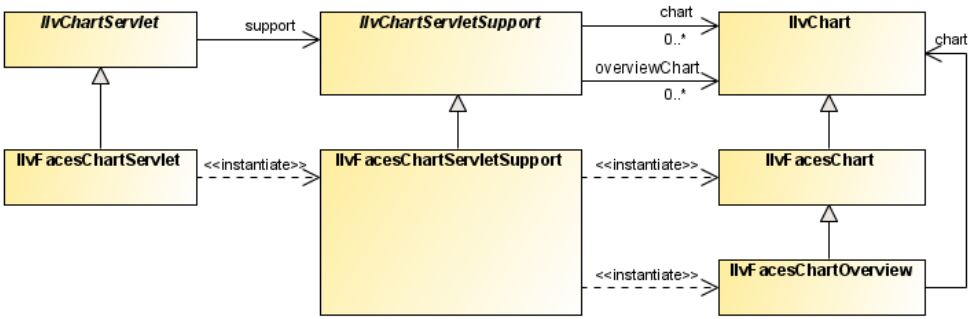
JSF Components in JViews Charts uses servlet technology to produce the images that are the visual representation of the component on the client side.



Faces view components



Faces interactors and other components



Faces servlet

Dedicated servlet, servlet support, and components are available to help create an application, see *Servlet and component classes*.

Servlet and component classes

Name	Description
`ilog.views.chart.faces.servlet.IlvFacesChart`	A dedicated `IlvChart` component.
`ilog.views.chart.faces.servlet.IlvFacesChartLegend`	A dedicated `IlvLegend` component.
`ilog.views.chart.faces.servlet.IlvFacesChartOverview`	A dedicated chart overview that extends `IlvChart`.
`ilog.views.chart.faces.servlet.IlvFacesChartServlet`	A dedicated `IlvChartServlet`.
`ilog.views.chart.faces.servlet.IlvFacesChartServletSupport`	A dedicated `IlvChartServletSupport`.
`ilog.views.chart.faces.dhtml.component.IlvChartDHTMLView`	A view component, extended to have DHTML rendering that displays an `IlvFacesChart`.
`ilog.views.chart.faces.dhtml.component.IlvChartDHTMLOverview`	An overview component, extended to have DHTML rendering that displays an `IlvFacesChartOverview`.
`ilog.views.chart.faces.dhtml.component.IlvChartDHTMLLegend`	A legend component, extended to have DHTML rendering that displays an `IlvFacesChartLegend`.
`ilog.views.chart.faces.interactor.IlvChartPanInteractor`	An interactor that allows you to pan the view.
`ilog.views.chart.faces.interactor.IlvChartPickInteractor`	An interactor that allows you to execute an action in the servlet context by clicking the image.
`ilog.views.chart.faces.interactor.IlvChartZoomInteractor`	An interactor that allows you to zoom the view.
`ilog.views.faces.component.IlvFacesContextualMenu`	A contextual popup menu.

For more information about DHTML architecture, see *DHTML thin-client support in JViews Framework* DHTML Thin-Client Support.

The JViews Charts Faces component set

Presents some examples to illustrate how to use the JViews Charts Faces components

In this section

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

Charts designer project
Presents an example using the Designer for JViews Charts.

Data source binding in JViews Charts
Presents examples of connecting data source components to chart components.

Styling chart data with CSS
Describes how to customize the chart data display using Cascading Style Sheets.

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

Installing interactors in a chart
Describes how to install interactors in a chart.

Connecting a chart view to a message box
Describes how to connect a message box to a `chartView`.

Setting the overview
Describes how to attach an overview to a `chartView`.

The legend component
Describes how to connect a legend component to a `chartView`.

Mixing with standard JavaServer Faces components
Describes how to configure and access a `chartView` component using standard JavaServer Faces components.

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.

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

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 chart view

The first and simplest page that can be made with a JViews Charts Faces component is a chart view showing the default built-in data set.

Creating an empty view

To specify an empty view:

```
<jvcf:chartView style="width:500px; height:300px;" />
```

This produces a 500 by 300 pixel chart.

Declaring the namespace

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

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

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.

```
<jvcf:chartView width="500" height="500" />
```

Charts designer project

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

For more information about the Designer for JViews Charts, see Using the Designer.

```
<jvcf:chartView id="chart"
                data="data/chart.icpr"
                style="width:800;height:400" />
```

You can set the CSS stylesheet and data source separately.

Data source binding in JViews Charts

If a project is not already set and you want to set a data source to a chart, a data source component should be connected to the chart component in order to display something.

Using an XML file

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

```
<jvcf:chartView style="width:500 px; height:300 px;"
                data="resources/data.xml" />
```

Using a value binding

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

```
<jvcf:chartView [...] data="#{dataBean.dataSource}" />
```

The bean should then provide the data source through its `getDataSource` method:

```
 public IlvDataSource getDataSource() {
    if(source == null) {
      IlvDataSource source = new IlvDefaultDataSource();
      double x = {1, 3, 2, 4, 6, 5};
      IlvDefaultDataSet dds = new IlvDefaultDataSet("Sample", x);
      source.setDataSet(0, dds);
    }
    return source;
}
```

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

```
<faces-config>
  <managed-bean>
    <description> A Data Bean </description>
    <managed-bean-name>dataBean</managed-bean-name>
    <managed-bean-class>DataBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

For further information about these configuration files, see the *JavaServer Faces* specifications.

Creating a component in a managed bean

Another way to specify the data source is to create an `IlvFacesChart` component directly in a managed bean:

```
<jvcf:chartView id="chart4"
                style="width:500;height:300;"
                chart="#{chartBean.chart}" />
```

Here the `IlvFacesChart` component is created directly by your bean instance and you can set the data source in the bean code as shown in the bean getter:

```
public IlvChart getChart() {
    try {
      IlvFacesChart chart = new IlvFacesChart();
      IlvDataSource source = new IlvDefaultDataSource();
      double x[] = {1, 3, 2, 4, 6, 5, 10, 2, 3, 0};
      IlvDefaultDataSet dds = new IlvDefaultDataSet("Sample", x);
      source.setDataSet(0, dds);
      chart.setDataSource(source);
      return chart;
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }
```

The bean must be declared in the `faces-config.xml` or the `managed-bean.xml` file.

Styling chart data 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:

```
<jvcf:chartView id="chart5" [...]  styleSheets="data/styleSheet.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.

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 `IlvFacesChartImageMapGenerator` and `IlvIMapDefinition` and the associated sample **Bank Account** for details of how to implement an image map object in JViews Charts.

JViews Charts

Adding and displaying an image map

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

```
<jvcf:chartView [..] generateImageMap="true"
imageMapGenerator="#{chartBean.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.

```
<jvcf:chartView [..] 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.

Installing interactors in a chart

You can install interactors in the chart to allow interaction with the chart. For example, install a zoom interactor:

```
<jvcf:chartView [...]  interactorId="zoom" />
<jvcf:chartZoomInteractor id="zoom" />
```

You can zoom on the chart by clicking and dragging a rectangle. By default, the zoom interactor only zooms along the x-axis. To zoom the chart freely or to constrain it along the y-axis, use the `XZoomAllowed` and `YZoomAllowed` attributes. You can also customize the appearance of the zoom interactor rectangle by using the `lineWidth` and `lineColor` attributes. A pan interactor is also available to scroll a `chartView`. A message box component can also be used to display messages originating from interactors and other components.

Connecting a chart view to a message box

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

```
<jvcf:chartView [...]  messageBoxId="messageBox"/>
<jv:messageBox id="messageBox" [...]  />
```

The messages issued are now displayed in the message box.

Setting the overview

To assist navigation in a zoomed chart, you can attach an overview to a `chartView`. The overview shows in a rectangle the visible part of the data displayed in the `chartView`. To connect the overview to the `chartView`, you must use the `viewId` attribute of the overview:

```
<jvcf:chartView id="chart8" [...] />
<jvcf:chartOverview [...] viewId="chart8" />
```

The overview can also be customized by using the `lineWidth` or `lineColor` attributes.

The legend component

A legend component is available to display the chart legend in a separate component:

```
<jvcf:chartLegend [...]  viewId="chartView" [...] />
```

This component is connected to the main chart through the `viewId` attribute.

Mixing with standard JavaServer Faces components

You can also use standard JavaServer Faces components to configure and access your `chartView` component. Here is an example of how to use an input text component to configure the header and footer values of a `chartView`.

The following code uses the `chart` attribute value binding, so that the `valueChangeListener` of an `inputText` can easily access the header of the `IlvChart`:

```
<jvcf:chartView [...] chart="#{chartBean.chart}" [...]  />
<h:inputText value="#{chartBean.header}"
             valueChangeListener="#{chartBean.setHeaderLabel}"/>
```

Here is the corresponding Java code:

```
public void setHeaderLabel(ValueChangeEvent evt) {
    chart.setHeaderText((String) evt.getNewValue());
}

public Object getHeader() {
    return header;
}

public void setHeader(Object object) {
    header = object;
}
```

When issuing a submit request, the value change listener is called and changes the header of the chart accordingly to the text entered in the `inputText` component.

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 Charts.

```
<jvcf:chartView [...]>
  <jvcf:chartContextualMenu [...]/>
</jvcf:chartView/>
```

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 Charts use the `menu`, `menuItem`, and `menuSeparator` tags as in the following example.

```
      <jvcf:chartContextualMenu>
        <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.zoomInX()"
                       image="images/zoom.gif" />
          <jv:menuItem label="Zoom Out"
                       onclick="viewID.zoomOutX()"
                       image="images/unzoom.gif"/>
          <jv:menuItem label="Zoom to Fit"
                       onclick="viewID.zoomToFit()"
                       image="images/zoomfit.gif"/>
          <jv:menuSeparator/>
          <jv:menuItem label="Select"
                       actionListener="#{chartBean.action}"
                       image="images/arrow.gif"
                       invocationContext="IMAGE_SERVLET_CONTEXT" />
        </jv:menu>
      </jvcf:chartContextualMenu>
```

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 Charts

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

```
<jvcf:chartContextualMenu factory="#{bean.factory}" />
 <jvcf:chartContextualMenu 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 Charts

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

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 Charts

```
<jvcf: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.

JavaScript objects

Each time a JViews Charts 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:

```
<jvcf:chartView id="chart9" [...] />
```

will be rendered as the following JavaScript code:

```
var chart9 = new IlvChartViewProxy ('chart9',' ...);
```

You can modify the object locally by using a set of methods attached to this object. For further information about available JavaScript objects, see `Javascript API`.

For example, the following code defines two buttons that install respectively a zoom interactor and a pan interactor on a `chartView` .

```
<jvf:imageButton [...] onclick="chart9.setInteractor(zoomInteractor)"  />
<jvf:imageButton [...] onclick="chart9.setInteractor(panInteractor)" />
<jvcf:chartView id="chart9" [...] />
```

At rendering time, an `IlvChartViewProxy` JavaScript object is created that is accessible through the `chart` JavaScript variable. Then, since `zoomInteractor` and `panInteractor` JavaScript objects have been created in the same way, you can directly set one of these interactors 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.

Contexts for actions on the view

Describes the contexts in which actions can be executed in response to interactions on the view.

In this section

Introduction
Describes the JavaServer Faces lifecycle and image servlet contexts for actions on the view.

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.

Introduction

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 Charts

To highlight a point in a chart view, a chart select interactor must be installed on the chart view. The `value` property of the interactor holds the `IlvDataSetPoint` that was clicked. Thus, a `valueChangeListener` can be registered to handle the selection event.

Installing a chart select interactor and a listener

```
<jvcf:chartSelectInteractor id="selectInteractor"
                  valueChangeListener="#{demoBean.pointSelected}"

                  pickingMode="item"
                              invocationContext="JSF_CONTEXT">
<jvcf:chartView id="chart" 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 pointSelected(ValueChangeEvent evt) {
  IlvDataSetPoint point = (IlvDataSetPoint) evt.getNewValue();

  if (point != null) {

              //The source of the event is the interactor
    IlvObjectSelectInteractor interactor =
                  (IlvObjectSelectInteractor) evt.getSource();
              //Retrieve the JSF view connected to the interactor
     IlvChartDHTMLView jsfView = (IlvChartDHTMLView) interactor.getView();

              //Retrieve the IlvChart wrapped by the JSF component.
    IlvChart chart = jsfView.getChart();
    //Set a pseudo class on the display point.
    //A CSS rule like point:selected { ... }
                   //will customize the graphic representation of the point.

    chart.setPseudoClasses(point.getDataSet(),
                           point.getIndex(),
                           new String[]{"selected"} );
```

```
}
```

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 Charts)

```
<jvcf:chartSelectInteractor id="selectInteractor"
                  valueChangeListener="#{demoBean.pointSelected}"

invocationContext="IMAGE_SERVLET_CONTEXT">
                  pickingMode="item"
<jvcf:chartView id="chart" 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 Charts:

- 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 `IlvChart` object as shown in *Java code of the 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 Charts

Using JavaScript actions in a JSP page

```
[...]
<jvcf:chartZoomInteractor id="zoom" [...] />
<jv:imageButton onclick="#{chartBean.setZoomAction}"/>
<jvcf:chartView id="chart" [...] />
[...]
```

Using JavaScript actions in a managed bean

```
public class ChartBean {
[...]
  private String setZoomAction;
  public ChartBean(){
    setZoomAction =
      IlvFacesUtil.encodeJavaScriptVariables("${chart}.setInteractor(${
        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 Charts

```
<jvcf chartView [...] servlet=
    "ilog.views.chart.faces.dhtml.servlet.IlvFacesChartServlet />"
```

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*.

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.

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">chart.updateImage();</script>
```

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

For tips and tricks on how to push JViews components, look at the push example installed with JViews Charts at ***<install-dir>*** **`/jviews-charts8.6/codefragments/jsf-charts-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.

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 Charts application examples configured for use with Facelets or Trinidad, see ***<install-dir>*`/jviews-charts8.6/codefragments/jsf-chart-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" />
```

For JViews Charts, the contextual menu element is `chartContextualMenu`.

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 an application as a DHTML-only thin client

Describes how to deploy an 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 architecture
Describes the thin client framework provided with the JViews Charts library.

Chart servlet package
Identifies the location of the classes used to build thin-client applications.

The IlvChartServlet class
The JViews Charts library provides a predefined lightweight servlet abstract class named `IlvChartServlet`. This class is a subclass of the `HttpServlet` class from the Java servlet API that automatically delegates the requests handling to an instance of `IlvChartServletSupport`.

The IlvChartServletSupport class
Describes the use of the `IlvChartServletSupport` class in handling HTTP requests to generate an image or an image map.

Server actions
Describes the definition of server-side actions using JViews Charts thin-client features.

Adding support for custom image formats
Describes how to add support for image formats other than JPEG and PNG.

Choosing the multithreading mode
Describes multithreading modes for JViews Charts components.

Writing a basic server side application
This section shows you the basic steps needed to create a simple server-side application that loads data from a chart, and sends the images in response to a client request.

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 Charts 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 architecture

The thin client framework provided with the JViews Charts library is based on the standard Java™ Servlet technology. For more information on the Java Servlet technology, you can visit the JavaSoft site at *http://java.sun.com/products/servlet*.

Publishing charts graphical representations of a data model through a web server typically consists of using the JViews Charts library on the server side to build and handle the data model and to generate the corresponding chart displays. Basically, this is performed by means of a servlet that answers to the HTTP requests sent by a client, and delivers the display of the charts to the client.

The default display outputs supported by the JViews Charts library are JPEG and PNG images. In addition to this image output, a corresponding client-side image map can also be generated to allow the user to add his own client-side interaction code.

Chart servlet package

All the classes needed to build thin-client applications are contained in the `ilog.views.chart.servlet` package. The core classes of the framework are the abstract `IlvChartServlet` and `IlvChartServletSupport` classes. These classes are responsible for handling the HTTP requests received, and for sending the expected response according to the request parameters.

Developing a server-side application for handling charts generation consists of creating a servlet that produces an image of a chart to the client.

The image and image maps generation are performed by the `IlvChartServletSupport` class according to the request parameters. The generation of the server-side charts consists of propagating the HTTP requests received by your servlet to an `IlvChartServletSupport` instance. This is typically implemented in the `doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` (or `doPost(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)`) method of an `HttpServlet` subclass, like in the following code:

```
public void doGet(HttpServletRequest request,
                              HttpServletResponse response)
 {
    if (!getServletSupport().handleRequest(request, response)) {
          // Handle requests other than chart image generation
          doSomething(request, response);
    }
 }
```

Invoking the servlet support `handleRequest(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method is automatically performed by the `IlvChartServlet` class. This class can be used as the basis to develop your servlets, if you are not adding server-side functionalities to an existing application.

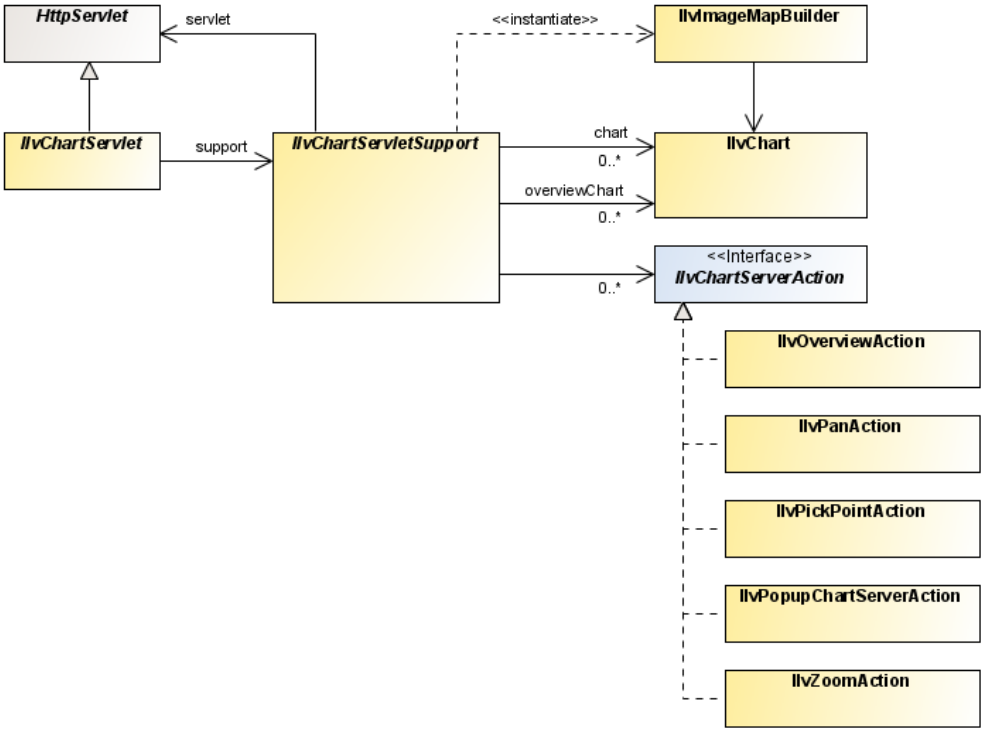
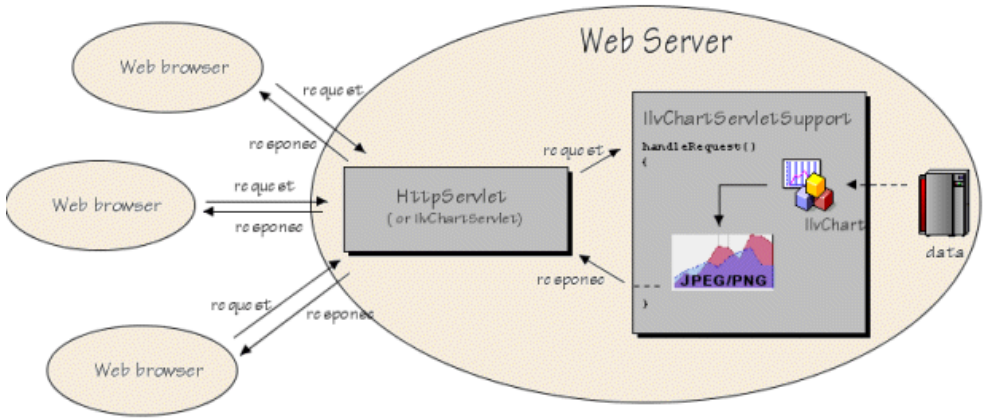


Chart servlet classes relationships

The following figure shows you how the mechanism works:



The IlvChartServlet class

The JViews Charts library provides a predefined lightweight servlet abstract class named `IlvChartServlet`. This class is a subclass of the `HttpServlet` class from the Java servlet API that automatically delegates the requests handling to an instance of `IlvChartServletSupport`.

In this section

Creating an IlvChartServletSupport
Describes the creation of an `IlvChartServletSupport` subclass.

Handling sessions
Describes how to handle HTTP sessions using the `IlvChartServlet` class.

Creating an IlvChartServletSupport

This instance is created by the `IlvChartServlet` when the first request is received. Since the `IlvChartServletSupport` class is abstract and should be subclassed, the instance is created by the abstract `createServletSupport()` factory method, which has to be overridden to return an instance of your own `IlvChartServletSupport` subclass.

Handling sessions

The `IlvChartServlet` class supports the HTTP session concept by means of the `prepareSession(javax.servlet.http.HttpServletRequest)` method. This method is called at every request to allow the user to prepare and configure a session for his servlet.

Assume a servlet only uses one `IlvChart` instance shared between all the clients. If this architecture works well in a read-only context, it does not work anymore if the data model can be modified by the clients. Indeed, any modification of the chart data model made by a client would be seen by all the clients. A solution is to create a session for every request. You can then create one `IlvChart` connected to a specific data model and store it as a parameter of the session. In this way, a chart is created for each session opened by a client instead of being created for all the clients.

Another aspect to consider when you work with sessions is memory leak. When you bind a chart as an attribute of an `HttpSession`, proper cleanup of the chart must occur when the session is invalidated or expires. This allows the chart to be garbage collected or restored to a pool of available charts, depending on your choice of strategies. Instead of binding the chart directly to the session, use an instance of the `IlvChartSessionAttribute` as a proxy.

The following code extract shows how to use it:

```
IlvChart chart = null;
HttpSession session = request.getSession();
if (session.isNew()) {
   chart = ... create new chart or fetch from pool ...
   IlvChartSessionAttribute proxy = new IlvChartSessionAttribute(chart);
   session.setAttribute(CHART_KEY, proxy);
} else {
   IlvChartSessionAttribute proxy =
     (IlvChartSessionAttribute)session.getAttribute(CHART_KEY);
   if (proxy != null)
     chart = proxy.getChart();
}
if (chart == null)
   throw new ServletException("session problem");
```

The IlvChartServletSupport class

Describes the use of the `IlvChartServletSupport` class in handling HTTP requests to generate an image or an image map.

In this section

Overview
Introduces the `IlcChartServletSupport` class.

The image request
Describes the syntax and parameters of an image request.

The image map request
Describes the syntax and parameters of an image map request and the process for generating or customizing image maps..

Overview

The `IlvChartServletSupport` class responds to HTTP requests to generate an image from an `IlvChart`. This class allows you to add the functionalities of the JViews Charts Servlet framework into your existing servlets by invoking the `handleRequest(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method from your servlet `doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method.

The servlet support can respond to two different types of HTTP requests:

- image request
- image map request

The image request

The image request produces either a JPEG or PNG image from the `IlvChart`. The syntax is:

```
myservlet?request=image
                         &width=width of the returned image
                         &height=height of the returned image
                         &format=output format of the returned image
                         [&bgcolor=image background color]
                         [&action=name of a server action]
                         [&comp=the chart component to dump]
```

The following table contains a detailed description of the parameters:

Parameter name	Parameter value	Description
`request`	image	Asks the servlet to generate an image.
`width`	Integer	Width of the image.
`height`	Integer	Height of the image.
`format`	“JPEG” “PNG”	Format of the resulting image.
`comp`	“chart” “area” “legend”	The component to dump.
`bgcolor`	An hexadecimal color value as “0xRRGGBB”	The background color of the image.
`action`	String	The action name to be executed on the servlet.

For example, the following request will produce an image of size (300, 200) of an `IlvChart` component as a PNG image:

```
myservlet?request=image&width=300&height=200&comp=chart&format=PNG
```

The image map request

The image map request produces a client-side image map associated with an image. The image can be:

- Explicitly specified in the request (as a URL),
- Dynamically generated by the servlet. In this case, you should specify the parameters for the image request in addition to the image map request parameters.

The syntax of the image map request is:

```
myservlet?request=imagemap
          &width=width of the image
          &height=height of the image
          [&mapname=imagemap]
          {[&image=URL of an image] | {
           [&format=output format of the image]
           [&comp=chart]
           [&bgcolor=image background color]
           [&action=name of a server action]}}
```

The following table contains a detailed description of the parameters:

Parameter name	Parameter value	Description
`request`	"imagemap"	Ask the servlet to generate an image map.
`width`	Integer	Width of the image.
`height`	Integer	Height of the image.
`mapname`	String	The name of the map. The default value is "imagemap".
`image`	String	The URL of an associated image. If this parameter is not specified, then an image is automatically generated.
`format`	"JPEG", "PNG"	Format of the image automatically generated. This parameter is required only if no image parameter has been specified.
`comp`	"chart", "area", "legend"	The component to dump. This optional parameter is only taken into account if no image parameter has been specified. The default value is "chart".
`bgcolor`	An hexadecimal color value as "0xRRGGBB"	The background color of the image.
`action`	String	The action name to be executed on the servlet.

For example, the following request asks for the servlet to generate both an image and an image map:

```
http://host/servlet/myservlet?request=imagemap
```

```
                                &width=400
                                &height=200
                                &format=JPEG
```

The resulting HTML code looks like:

```
<map name="imagemap">
<area shape="poly" coords="364,165 [...]" href="..." alt="..." title="...”>
...
</map>
<img usemap="#imagemap" width="400" height="330" src="/servlet/
myservlet?request=image&height=330&width=400&format=JPEG"" border=0>
```

The call generates an HTML code fragment containing a client-side image map definition and an image. The contents of the image is then generated by another call to the servlet by means of an image request.

Generating an image map

Image maps are images with an attached map that points out certain hot spots, or clickable areas. In the JViews Charts library, a clickable area can be generated for specific data points or whole data sets.

The image map generation is handled by the `IlvChartServletSupport` class, and performed by means of an instance of the `IlvImageMapBuilder` class. This class is responsible for generating the area tags of the image map according to a specified configuration. The `IlvImageMapBuilder` instance used during the image map generation is retrieved with the `createImageMapBuilder(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)` factory method.

Area tags are generated according to a configuration that is defined by an instance of a concrete implementation of the `IlvIMapDefinition` abstract class. A default implementation of this class is provided by means of the `IlvDefaultIMapDefinition` class.

Warning: Since ILOG JViews 5.5, the `IlvIMapDefinition` interface has been modified to extend the current image map support to `IlvLegend` components. Implementations based on the JViews 5.0 API must be modified to implement the new `IlvIMapDefintion.getAttributes(IlvLegendItem)` method.

The map definition instance is retrieved by invoking the `getIMapDefinition(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)` method. By default, this method returns `null` leading to an empty image map area. You should override this method to return an `IlvIMapDefinition` instance configured with the correct attribute values of the area tags.

The `IlvIMapDefinition` abstract class also determines what the area tags represent: a whole data set or specific data points. This is called the *map definition type*, and is specified at construction time as a parameter to the `IlvIMapDefinition` constructor. This type can be retrieved using the `getType()` method. A data points-based image map has the `POINT_MAP` type and a data set-based image map has the `DATASET_MAP` type.

By default, the generated attributes of the area tags are SHAPE, COORDS, HREF, TITLE and ALT.

Area tags attributes are handled through IlvIMapAttributes instances. This interface defines the necessary methods to fetch the HREF, TITLE, ALT, and TARGET attributes values, the SHAPE and COORDS attributes values being automatically generated. A default implementation is provided by means of the IlvDefaultIMapAttributes class. This class handles HREF and TITLE values separately (both can be null), and maps the values of the ALT attribute to the values of the TITLE attribute.

The following example shows an IlvChartServletSupport subclass that overrides the getIMapDefinition(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart) method to return an IlvDefaultIMapDefinition instance configured with the POINT_MAP type and with specific HREF values for the HREF attributes:

```
protected IlvIMapDefinition getIMapDefinition(HttpServletRequest request,
                                              IlvChart chart)
{
    IlvIMapDefinition mapdef = null;
    IlvDataSet dataSet = ...; // The data set for which we want map areas.
    List hrefs = ...; // The list of href for the map area.
    String[] hrefsValues = new String[hrefs.size()];
    hrefsValues = (String[])hrefs.toArray(hrefsValues);

    // Create attributes initialized with the contents of the hrefs list.
    IlvIMapAttributes[] attrs = IlvDefaultIMapAttributes.create(hrefsValues);

    try {
        // Create a map definition that associates the data set with the
        // area attributes.
        mapdef =
              new IlvDefaultIMapDefinition(new IlvDataSet[] {dataSet},
                                           new IlvIMapAttributes[][]
{attrs});
    } catch (IllegalArgumentException e) {
        System.err.println("Cannot create map definition :" + e.getMessage())
;
    }
    return mapdef;
    }
```

Customizing image maps

The IlvChartServletSupport class allows you to customize the image generation by means of three methods:

- createImageMapBuilder(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)

  This method allows you to return an instance of your own IlvImageMapBuilder subclass. This class is responsible for generating the image map HTML code fragment, that is:

  - The area tags corresponding to the renderers, by means of the getRendererTags(ilog.views.chart.servlet.IlvIMapDefinition) method.

- The area tag corresponding to the chart header, by means of the `getHeaderTag(ilog.views.chart.servlet.IlvIMapAttributes)` method.
- The area tag corresponding to the chart footer, by means of the `getFooterTag(ilog.views.chart.servlet.IlvIMapAttributes)` method.

Each of these methods can be overridden if you need to perform additional actions.

♦ `getIMapDefinition(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)`

This method should be overridden to return an `IlvIMapDefinition` instance configured with area attributes.

♦ `generateMapAreaTags(java.io.PrintWriter, ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)`

This method is the core of the image map generation process. The default implementation first creates an `IlvImageMapBuilder` instance invoking `createImageMapBuilder`. Then, it generates the renderer area tags calling the `IlvImageMapBuilder.getRendererTags` method, passing the `IlvIMapDefinition` instance returned by `getIMapDefinition` as parameter.

Note: The default implementation does not generate the header and footer tags. You have to override the `generateMapAreaTags` method and explicitly call `IlvImageMapBuilder.getFooterTag` and `IlvImageMapBuilder.getHeaderTag` methods.

You can find a complete example showing the use of image map in **<installdir>/jviews-charts86/samples/servlet-chart-imgmap/srchtml/imgmap/ChartImageMap.java**.

Server actions

The JViews Charts thin-client support gives you a simplified way to define new actions that should take place on the server side. For example, suppose you want to allow the user to change the legend visibility from its browser. Changing the legend visibility must be done on the server side before a new image is generated. The notion of a "server-side action" exists to perform such behavior. An action is defined by a name and a set of string parameters.

On the client side, you tell the server to execute an action by sending an image request with the “action” parameter set. The value of this parameter is the action name concatenated with an in-parenthesis-comma-separated list of the required parameters. For example, the following request executes the `LegendVisibilityAction` action with the expected visibility as a parameter:

On the server side, you detect that an action was requested and you execute the action before the image is generated by implementing the `IlvChartServerAction` interface. To listen for an action request from the client and execute the action on the server side, you register the action with your instance of `IlvChartServletSupport` using the `addServerAction` method.

Server actions can be executed at two different times: either when the request has been just received before anything else, or just before the image is generated. In the first case, the actions are executed in the request thread, allowing you to execute actions that would typically be time-consuming and/or do not modify the visual appearance of the chart. In the second case, the actions are executed in the event dispatch thread. You specify in which thread an action should be executed by implementing the `IlvChartServerAction.getExecutingThread` method.

For example, the corresponding implementation of the `LegendVisibilityAction` is:

```
class LegendVisibilityAction implements IlvChartServerAction
{
    /** The action name. */
    static final String ACTION_NAME    = "LegendVisibilityAction";

    public void actionPerformed(IlvChartServerActionEvent event)
      throws ServletException
    {
       boolean visible =
            Boolean.valueOf(
                      event.getParameters()[0]).booleanValue();
       event.getChart().setLegendVisible(visible);
    }

    public int getExecutionThread()
    {
        return IlvChartServerAction.SWING_EVENT_THREAD;
    }
}
```

Adding support for custom image formats

The generation of the image in a specific format is performed by means of the image *encoders*. The `IlvChartServletSupport` class defines the image encoder objects through the `IlvImageEncoder` interface. This interface defines the behavior of a class that can encode an image onto an `HttpServletResponse`. To generate JPEG and PNG images the JViews Charts library provides two implementations of the `IlvImageEncoder` interface: the `IlvJPEGEncoder` and `IlvPNGEncoder` classes.

To add support for another image format, follow these steps:

1. Write a new implementation of the `IlvImageEncoder` interface that encodes an image in the given output format.
2. Register it on the `IlvChartServletSupport` class so that the new format can be recognized as a supported format. Registering a new image encoder is done by means of the following method: `setImageEncoder(java.lang.String, ilog.views.chart.servlet.IlvImageEncoder)`.

Choosing the multithreading mode

Charts components can be used in two modes, regarding multithreading. They differ in their choice of which thread to use for executing the `paint()` operations.

The *using event thread* mode is suitable for Swing GUI applications. It causes all drawing and all Swing API accesses to be performed in the AWT EventQueue thread. This is a Swing requirement.

The *current thread* mode causes the drawing to be performed in the caller thread, regardless whether it is the AWT EventQueue thread or not.

The *current thread* mode has some advantages for server-side processing, namely:

- It increases performance, especially on multiprocessor machines, to process every request in its originating thread. (The AWT-Event thread can be executed on only one processor at any time.)
- Better interoperability with Tomcat. (Some versions of Tomcat 5 do not terminate properly upon request from the "shutdown" script when the AWT-Event thread has been in use.)

The drawback of the *current thread* mode is that it is unable to draw other JComponents than `IlvChart`, `IlvChart.Area`, `IlvLegend` and `IlvLegendItems`, except for header and footer: these can be drawn if they are `JLabel` instances.

To set the mode for all charts, use the static method `IlvChart.setNoEventThreadUpdate`.

If set to `true`, this method activates the *current thread* mode; if set to `false`, it activates the *using event thread* mode.

To set the mode for an individual chart, use the method `IlvChart.setUsingEventThread`.

If set to `true`, it activates the *using event thread* mode; if set to `false`, it activates the *current thread* mode.

The default in general is the *using event thread* mode. The default for JSF components is the *current thread* mode. The default for an individual chart is according to the general setting (`getNoEventThreadUpdate()`).

There are methods in the `IlvChart` class that can have two different behaviors, according to the mode: *using event thread* or *current thread* mode. These methods are:

using event thread mode	current thread mode
paint	paintCurrentThread
print	printCurrentThread
paintToFO	paintToFOCurrentThread
toImage	toImage
toPNG	toPNG

Other functionality, however, takes automatically into account the `isUsingEventThread()` result.

In any case, the `IlvChartServletSupport` base class handles all these threading issues for you. It ensures that all the requests to modify a chart and to generate its image are moved from the HTTP request thread onto the AWT event dispatch thread, as necessary. If you need to extend the basic functionalities and plug your own mechanism, you must consider this multithreading issue, and look at the services provided by the `IlvServletUtil` class.

Writing a basic server side application

This section shows you the basic steps needed to create a simple server-side application that loads data from a chart, and sends the images in response to a client request.

In this section

Example: The Basic Servlet
Presents an example of a servlet that loads data from a chart and sends images in response to a client request.

Installing and running the example
Describes the steps involved in building, installing, and viewing the example on a Tomcat server..

Implementing the server-side application
Presents an overview of the servlet example.

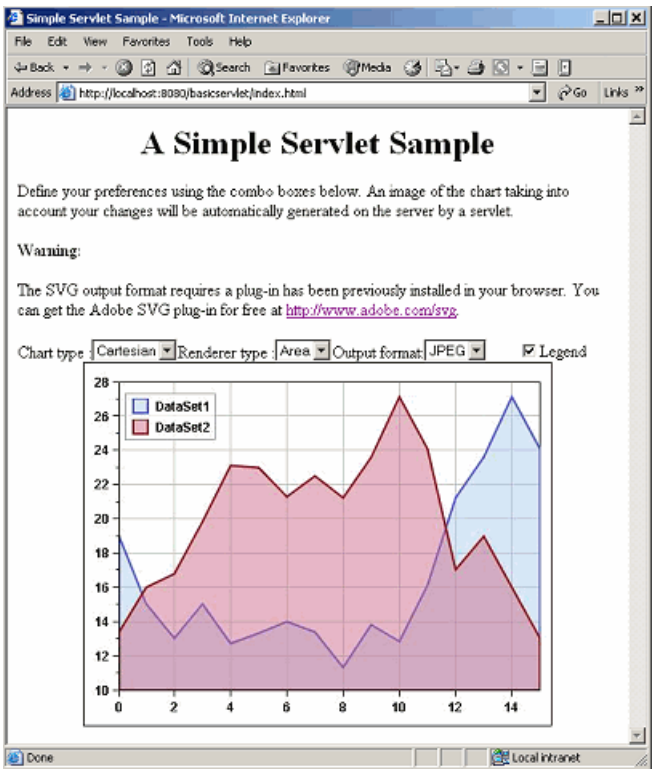
Creating the servlet
Presents the code to create the servlet.

Creating the servlet support
Identifies the parameters of the servlet request and describes how to create the servlet support.

Example: The Basic Servlet

The Basic Serlvet example can be found in the `<installdir>/jviews-charts86/samples/servlet-chart-basic/srchtml/basicservlet/` directory.

This example shows you the various chart representations of the same data model according to the user's preferences set on the client:



A Simple Servlet example

The example offers the following possibilities:

- The chart can be either Cartesian or polar.
- The data model graphical representation can be chosen among area, polyline, bar, and stairs.
- The output image format can be chosen among JPEG, PNG and SVG.
- The legend can be either visible or hidden.

The example is composed of the following files:

- The servlet that produces the images of chart representations of data loaded from an XML file.

<installdir>/jviews-charts86/samples/servlet-chart-basic/srchtml/basicservlet/BasicServlet.java

- The starting HTML file.

  <installdir>/jviews-charts86/samples/servlet-chart-basic/index.html

- The data file.

  <installdir>/jviews-charts86/samples/servlet-chart-basic/webpages/data.xml

Installing and running the example

To be able to run, this example requires a Web server that supports servlets. The example contains an `ant` script that allows you to easily build and install the example on a Tomcat server, the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.

Here are the steps for running the example:

1. Go to the IBM ILOG JViews directory

   ```
   <installdir>/jviews-charts86/samples/servlet-chart-basic
   ```

2. Execute the `ant` run entry:

   ```
   ant run
   ```

3. Once the server is up and running, launch a Web browser and open the example page:

   ```
   http://localhost:8080/basicservlet/index.html
   ```

Implementing the server-side application

A typical server-side chart application is composed of two main parts:

- The application itself, which holds the data model(s) and the chart(s).
- A servlet, which produces images to the client.

  Actually the servlet part consists of a servlet that handles HTTP requests, and of an associated `IlvChartServletSupport` instance that handles the chart image generation.

To simplify the example, the application part is actually bundled in our `IlvChartServletSupport` subclass that produces the images.

Creating the servlet

The servlet is the key element in a server-side application: it handles the requests sent by the clients, and builds an appropriate response depending on the request type, delegating the response to produce to an `IlvChartServletSupport` instance in the case of a chart image request.

When you write a server-side application with the JViews Charts library, you have the choice of using either your own `javax.servlet.http.HttpServlet` subclass (but in this case you have to write the code required to delegate the chart image generation to an `IlvChartServletSupport` instance), or the predefined and lightweight abstract `IlvChartServlet` class (that automatically delegates the image generation to an `IlvChartServletSupport` instance).

Since we do not want to add the chart generation capability to an existing application, we will use directly an `IlvChartServlet` instance. Here is the code of our servlet:

```
public class BasicServlet extends IlvChartServlet
{
    protected IlvChartServletSupport createServletSupport()
    {
       return new BasicSupport(getServletContext());
    }
}
```

Subclassing `IlvChartServlet` mainly consists of providing an implementation of the `createServletSupport()` abstract method. This method is responsible for creating the `IlvChartServletSupport` instance used by the servlet to generate the images. Our implementation creates a new `SimpleSupport` instance, passing the servlet context as parameter to the constructor.

Creating the servlet support

The `IlvChartServletSupport` class is the core class in the chart images generation process. Its `handleRequest(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method is automatically called by the servlet `doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method to handle all the incoming requests. Basically, this method calls the `prepareChart(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)` method (a hook method that can be used to perform some chart initialization job), and generates the image in the specified format. When subclassing `IlvChartServletSupport`, you have to override the `getChart(ilog.views.chart.servlet.IlvServletRequestParameters)` abstract method so that it returns the `IlvChart` object concerned by the request.

In our example, we want to be able to generate either a Cartesian or a polar chart, as well as to choose the graphical representation of our data model. These user's preferences will be set using the parameters of the servlet request.

These parameters are shown in the following table:

Parameter	Value
`chartType`	0 for Cartesian chart 1 for polar chart
`rendererType`	1 for polylines 2 for areas 3 for bars 4 for stairs

In addition to these new parameters, we also need to specify the output format of the image. We do not need to define a new parameter for this purpose because the `IlvChartServletSupport` already defines a format parameter to the image request. We just have to define a new parameter value ("SVG") for the SVG format.

We also want to allow the user to change the legend visibility. This will be done by means of a server action registered on the servlet. The action will take one parameter and set the legend visibility according to the parameter value.

To create the servlet support:

1. Subclass `IlvChartServletSupport` and define the various constants used:

```
class BasicSupport extends IlvChartServletSupport
{

    static final String DATAFILE = "data/data.xml";

    /** The parameter values. */
    static final String CHART_TYPE_PARAMETER  = "chartType";
    static final int CARTESIAN_CHART_TYPE     = 0;
    static final int POLAR_CHART_TYPE         = 1;
```

```
    static final String RENDERER_TYPE_PARAMETER   ="rendererType";
    static final int POLYLINE_TYPE            = 1;
    static final int AREA_TYPE                = 2;
    static final int BAR_TYPE                 = 3;
    static final int STAIR_TYPE               = 4;

    /** The SVG format parameter value. */
    static final String SVG_IMAGE_FORMAT      = "SVG";

    /** Rendering styles. */
    static BasicStroke stroke                 = new BasicStroke(2);
    static final IlvStyle[] styles            = {
         new IlvStyle(stroke, new Color(79,79,207), new Color
(175,207,239,128)),
        new IlvStyle(stroke, new Color(143,15,15), new Color(207,111,143,
 128))
    };

    /** The chart. */
    protected IlvChart chart;
```

To avoid creating an `IlvChart` instance for each request, a unique `IlvChart` instance will be used and held in the chart attribute.

2. Define the server action used to set the legend visibility:

```
static class LegendVisibilityAction implements IlvChartServerAction
{
    /** The action name. */
     static final String ACTION_NAME    = "ChangeLegendVisibilityAction";


    public void actionPerformed(IlvChartServerActionEvent event)
      throws ServletException
    {
        boolean visible =
               Boolean.valueOf(event.getParameters()[0]).booleanValue();

        event.getChart().setLegendVisible(visible);
    }
    public int getExecutionThread()
    {
        return IlvChartServerAction.SWING_EVENT_THREAD;
    }
}
```

The action parameter is retrieved from the `IlvChartServerActionEvent`, and the legend visibility is set accordingly.

3. Write the code required to initialize a new `SimpleSupport` instance:

```
public BasicSupport(ServletContext context)
```

```
{
    // Create the chart.
    chart = createChart(IlvChart.CARTESIAN);
    // Create the data source containing the values. Data is
    // read from an XML data file.
    IlvXMLDataSource dataSource = new IlvXMLDataSource();
    try {
        dataSource.load(context.getRealPath(DATAFILE),
                        new IlvXMLDataReader());
    } catch (Exception e) {
        e.printStackTrace();
    }
    // Set the current charts data source.
    chart.setDataSource(dataSource);
    // Initialize the renderer styles.
    chart.getRenderer(0).setStyles(styles);
}
```

Data is loaded from an XML data file using an `IlvXMLDataSource` instance that is directly connected to the chart. The `setDataSource(ilog.views.chart.data.IlvDataSource)` method automatically creates a chart renderer of the default type and associates it with the new data source.

4. Provide an implementation of the `getChart(ilog.views.chart.servlet.IlvServletRequestParameters)` method so that it returns the chart to dump. In our case, this method simply returns the chart attribute:

```
protected IlvChart getChart(HttpServletRequest request,
                            IlvServletRequestParameters params)
throws ServletException
{
    return chart;
}
```

5. Write the code that configures the chart and the renderers according to the user's preferences before generating the images. Preparing a chart before it is dumped as an image is the role of the `prepareChart(ilog.views.chart.servlet.IlvServletRequestParameters, ilog.views.chart.IlvChart)` method:

```
protected void prepareChart(HttpServletRequest request,
                            IlvChart chart)
throws ServletException
{
    IlvServletRequestParameters params =
        new IlvServletRequestParameters(request);
    int chartType = 0;
    try {
        chartType = params.getInteger(CHART_TYPE_PARAMETER);
    } catch (IlvParameterException pe) {
        throw new ServletException("Chart type parameter missing or
        badly initialized");
    }
    if (POLAR_CHART_TYPE == chartType) {
```

```
        chart.setType(IlvChart.POLAR);
        chart.getCoordinateSystem(0).setXCrossingValue(0);
    } else
        chart.setType(IlvChart.CARTESIAN);

    int type;
    try {
        type = params.getInteger(RENDERER_TYPE_PARAMETER);
    } catch (IlvParameterException pe) {
        throw new ServletException("Renderer type missing or badly
      initialized: ");
    }
    int rendererType;
    switch (type) {
       case POLYLINE_TYPE: rendererType = IlvChartRenderer.POLYLINE;break;

        case AREA_TYPE : rendererType = IlvChartRenderer.AREA;    break;

        case BAR_TYPE  : rendererType = IlvChartRenderer.BAR;     break;

        case STAIR_TYPE: rendererType = IlvChartRenderer.STAIR;   break;

        default:
            throw new ServletException("Unknown renderer type: " + type)
;
    }
    chart.setRendererType(0, rendererType);
    chart.getRenderer(0).setStyles(styles);
}
```

This method simply initializes the chart type according to the `chartType` request parameter, and sets the renderer type according to the `rendererType` request parameter.

6. Add the SVG generation.

   Because SVG is not part of the image formats natively supported, our `IlvChartServletSupport` subclass needs to handle the SVG generation by itself. This is done by means of a new method, `doGetSVG`, and by overriding the `IlvChartSerlvetSupport.handleRequest` method to call the `doGetSVG` method in the case of an SVG image request.

```
public boolean handleRequest(HttpServletRequest  request,
                             HttpServletResponse response)
 throws IOException, ServletException
{
   if (request.getParameter(IMAGE_FORMAT_PARAM).equals(SVG_IMAGE_FORMAT)
) {
       // Handle SVG generation.
       IlvServletRequestParameters params =
           new IlvServletRequestParameters(request);
       // Execute actions to be run in the request thread.
       executeAction(params, getChart(params),
                     IlvChartServerAction.REQUEST_THREAD);
```

```
        try {
            doGetSVG(response, params);
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
            return false;
        }
        return true;
    } else {
         return super.handleRequest(request, response);
    }
}
```

```
private void doGetSVG(HttpServletResponse response,
                          IlvServletRequestParameters params)
        throws IOException, ServletException, ParserConfigurationException

    {
         ...
         IlvServletRunnable svgPrinter = new IlvServletRunnable()
         {
             public void run() throws ServletException
             {
                 ...
                 prepareChart(httpParams, chart);
                   executeAction(httpParams, chart, IlvChartServerAction.
SWING_EVENT_THREAD);
                 ...
                 try {
                     // Draw the image.
                    Document document =
                         DocumentBuilderFactory.newInstance().
newDocumentBuilder().newDocument();
                     // Create an instance of the SVG generator.
                     SVGGraphics2D svgGenerator = new SVGGraphics2D
(document);
                        // Ask the test to render into the SVG Graphics2D
 implementation.
                     comp.paint(svgGenerator);
                     ...
                 } catch (Exception e) {
                     throw new ServletException(e.getMessage());
                 }
                 ...
              }
         };
         try {
             IlvServletUtil.invokeAndWait(svgPrinter);
         } catch (IOException x) {
             x.printStackTrace();
         }
    }
```

Since an `IlvChart` is a Swing component, painting should be done in the Swing event thread. To do so, we implement an `IlvServletRunnable` that generates the corresponding SVG code. This runnable is then passed to the `IlvServletUtil.invokeAndWait` method to be executed in the Swing event dispatch thread. The SVG generation is performed by painting the chart in an `SVGGraphics2D` paint context, a specialized Graphics context that generates SVG. This class is part of the Batik toolkit. More information about the Batik project can be found at *http://xml.apache.org/batik*.

DHTML thin-client support in JViews Framework

Describes the support for thin-client applications in JViews Framework.

In this section

Overview of thin-client support
Gives background information on the support for thin-client applications.

IBM® ILOG® JViews thin-client Web architecture
Describes how a thin-client application is structured.

Getting started with the IBM® ILOG® JViews thin client
Explains how to build the server and client sides of a thin-client application.

Installing and running the XML Grapher example
Explains how to install and run the XML Grapher example.

Developing the server
Describes the server side of a thin-client application and how to develop a server.

Developing the client
Describes the client side of a thin-client application and how to develop a dynamic HTML client by adding JavaScript™ components.

Adding client/server interactions
Describes how to add interactions between the server side and the client side.

Generating a client-side image map
Describes how to generate an image map on the client side.

The IlvManagerServlet class
Describes the predefined servlet and how to use it.

The IlvManagerServletSupport class
Describes how to add thin-client support to a servlet.

Controlling tiling
Describes how to control tiling on the client side and the server side.

Overview of thin-client support

The IBM® ILOG® JViews class library can be used on the client side where you develop Java™ applets or applications. It can also be used on the server side. Some Web browser applications require that the client stay very light, with most of the functionality residing in the server. The thin-client support in IBM® ILOG® JViews Framework allows you to create such applications easily. You can use the power of the IBM® ILOG® JViews class library to build complex two-dimensional representations on the Web server and use the Dynamic HTML thin-client support of your Web browser to display and interact with the images created by the server.

IBM® ILOG® JViews thin-client Web architecture

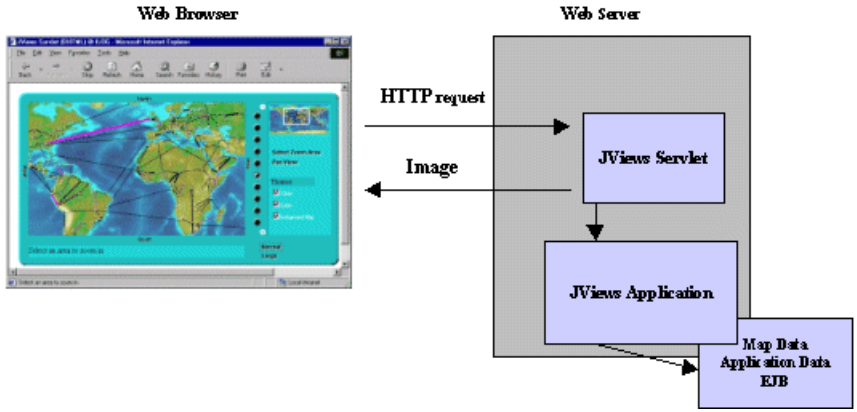
The IBM® ILOG® JViews thin-client support is based on the Java™ servlet technology. Servlets are Java programs that run on a web server. They act as a middle layer between HTTP requests coming from a Web browser or other HTTP clients such as applets or applications and the application or databases on the web server. The job of the servlet is to read and interpret HTTP requests coming from an HTTP client program and to generate a resulting document that in most cases is an HTML page.

For more information about servlet technology, you can visit the JavaSoft™ site *http://java.sun.com/products/servlet*.

You will also find their information about the web servers supporting Java servlets.

For the predefined types of IBM® ILOG® JViews clients, the content created by the servlet is primarily a JPEG image. On the client side, user interactions with the image are managed by code in Dynamic HTML scripts.

Creating a web application with IBM® ILOG® JViews consists of using the IBM® ILOG® JViews library on the server side to create complex two-dimensional displays based on application data that resides on the server. A servlet will answer HTTP requests from a client and deliver images to this client, as illustrated in the following figure.



Client-Server Display Interaction

IBM® ILOG® JViews Framework thin-client support contains the following:

- An abstract servlet class that can generate JPEG images from an IBM® ILOG® JViews display.
- A set of Dynamic HTML scripts written in JavaScript™ that will be used on the client side to display and interact with the image created on the server side.

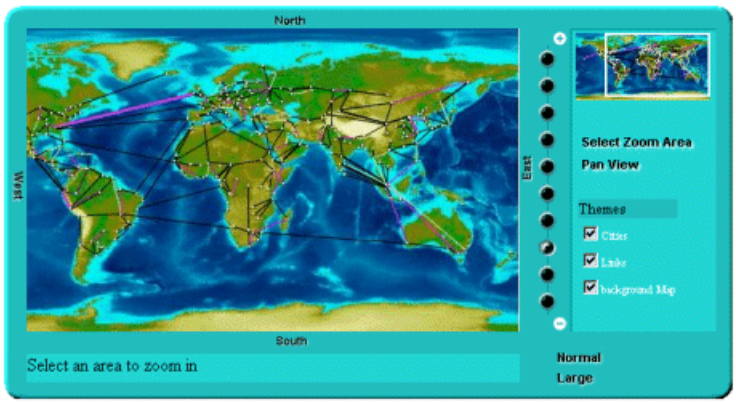
Creating an IBM® ILOG® JViews thin-client application consists of developing the server side and developing the client side.

Getting started with the IBM® ILOG® JViews thin client

The XML Grapher example shows how to build the server side and also how to create a Dynamic HTML client.

The XML Grapher example is available at **<*installdir*> /jviews-framework8.6/samples/xmlgrapher**.

This example allows you to display a network of interconnected cities on top of the map in a thin-client context.



The XML Grapher Example

The XML Grapher example is composed of the following pieces:

- An IBM® ILOG® JViews component that can read an XML file describing a set of interconnected cities and display them on top of a map as shown in the picture above.

  This component is located in the following files:

  <*installdir*> /jviews-framework86/samples/xmlgrapher/src/xmlgrapher/XmlGrapher.java

  <*installdir*> /jviews-framework86/samples/xmlgrapher/src/xmlgrapher/GrapherNode.java

- Some example XML files for the component, located in *<installdir>* / jviews-framework86/samples/xmlgrapher/webpages/data

- A servlet that can produce JPEG images from the component described above.

  The servlet is located in:

  <*installdir*> /jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java

- A Dynamic HTML client composed of:

  - The HTML starting page: **<*installdir*> /jviews-framework86/samples/xmlgrapher/webpages/dhtml/index.html**

- The set of JavaScript™ Dynamic HTML components, located in: ***<installdir>*** **/ jviews-framework86/lib/thinclient/javascript**
- Some images required for the example, located in: ***<installdir>*** **/ jviews-framework86/samples/xmlgrapher/webpages/dhtml/images**

Installing and running the XML Grapher example

This sample is compatible with the browsers and browser versions listed in the Release notes under *Requirements for running thin-client applications*. The example contains a WAR (Web ARchive) file that allows you to install the example easily on any server that supports the Servlet API 2.1 or later.

For your convenience, the WAR file has already been installed for you on the Apache Tomcat™ Web server that is supplied with the IBM® ILOG® JViews installation. Tomcat is the official reference implementation of the Servlet and JSP™ specifications. If you are already using an up-to-date Web or application server, there is a good chance that it already has everything you need. You can check the latest list of servers that support servlets at: *http://java.sun.com/products/servlet/industry.html*.

To be able to run, this example requires a Web server and a Web browser that supports Dynamic HTML (for the DHTML client).

To run the example on the TOMCAT web server supplied with the IBM® ILOG® JViews installation:

1. Set the JAVA_HOME environment variable to point to your Java™ Platform, Standard Edition installation.
2. Go to the TOMCAT `bin` directory located in

   ```
   <installdir>/jviews-framework86/tools/apache-tomcat-6.0.14/bin
   ```

3. Depending on your system, run the `startup.bat` or `startup.sh` script to run the Apache Tomcat™ server.
4. To see the example, launch a Web browser and open the page:

   http://localhost:8080/xmlgrapher/index.html

> **Note**: You must use `localhost` instead of the name of your machine. Otherwise, the sample applet may not be able to connect to the servlet.

The Web page gives you access to two different clients: a Dynamic HTML client and a thin Java client.

The IBM® ILOG® JViews servlets can run with the headless support that is built-in since Java SE 1.4, without an X server. For more information on this feature, refer to the Java SE *Release Notes*.

Developing the server

The server side of an IBM® ILOG® JViews thin-client application is composed of two main parts: the IBM® ILOG® JViews application itself, which can be any type of complex two-dimensional display built on top of the IBM® ILOG® JViews API, and a Servlet that produces JPEG images to the client.

The way the server side is built in the XML Grapher example helps in analyzing these parts.

The XML Grapher server

In the XML Grapher example, a graph of nodes and links is displayed on top of a map. This IBM® ILOG® JViews application is defined in the file `XmlGrapher.java`, located in ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java`**

> **Note**: This part of the example contains only standard IBM® ILOG® JViews code and is therefore not explained in detail. You will only see how the class is used to create the example. The application on the server side really depends on the type of information you want to display anyway.

The XmlGrapher class

The `XmlGrapher` class is a simple subclass of the IBM® ILOG® JViews `IlvManagerView` class.

The main functionality of this small component is to read an XML file describing nodes and links and to create an IBM® ILOG® JViews grapher that represents those nodes and links on top of a map. This is done in the method:

```
public void setNetwork(URL url)
```

The XML file contains information on the map and the bitmap file of the map. It contains a list of nodes, including the position, or location, of each node and information on links. In the example, the position, or location, is described by using x-y coordinates. In a real mapping application, the IBM® ILOG® JViews Maps API allows you to use geographical projections.

The `setNetwork` method parses the XML file, creates the map, and places the nodes and the links on top of the map. It also applies an orthogonal link layout algorithm to lay out the links automatically.

You can look at an XML example file in ***<install-dir>*** `/jviews-framework86/samples/xmlgrapher/webpages/data`.

The servlet

Once the application is built, you need to create a servlet that produces images of the application to a client. IBM® ILOG® JViews Framework provides a predefined servlet to

achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

The servlet created for the XML Grapher example is very simple. To understand in depth how the servlet works, read *The IlvManagerServlet class*. The servlet for the XML Grapher example is located in the file: ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java`** .

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;

import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  private XmlGrapher xmlGrapher;

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    xmlGrapher = new XmlGrapher();
    String xmlfile = config.getInitParameter("xmlfile");

    if (xmlfile == null) {
      xmlfile = config.getServletContext().getRealPath("/data/world.xml");
      xmlfile = "file:" + xmlfile;
    }
    try {
      xmlGrapher.setNetwork(new URL(xmlfile));
    } catch (MalformedURLException ex) {
    }
    setVerbose(true);
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                  IlvManagerView view)
  {
    return 30;
  }
```

```
}
```

The `import` statements:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

are required to use the Java Servlet API.

The `import` statements:

```
import ilog.views.*;
import ilog.views.servlet.*;
```

are required for using IBM® ILOG® JViews and the IBM® ILOG® JViews servlet support.

The `import` statement:

```
import demo.xmlgrapher.*;
```

is required for the XML Grapher class.

The `IlvManagerServlet`. class is an abstract Java™ class subclass of the `HTTPServlet` class from the Java servlet API. The `XmlGrapherServlet` inherits from the `IlvManagerServlet` class and defines only three methods.

The init method

This method initializes the servlet by creating an `XmlGrapher` object:

```
public void init(ServletConfig config) throws ServletException
{
    xmlGrapher = new XmlGrapher();
    ...
```

Then an XML file is read by the `XmlGrapher` object using the `setNetwork` method:

```
String xmlfile = config.getInitParameter("xmlfile");
if (xmlfile == null)
  xmlfile
       = config.getServletContext().
              getRealPath("/data/world.xml");

try {
  xmlGrapher.setNetwork(new URL("file:" + xmlfile));
} catch (MalformedURLException ex) {
}
```

The XML file can be specified in the configuration of the servlet. By default, the file `world.xml` is used.

The getManagerView method

The **getManagerView** method is the only abstract method of the `IlvManagerServlet` class and should return an `IlvManagerView` that will be used to generate the image. Here the `XmlGrapher` object is returned.

```
public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }
```

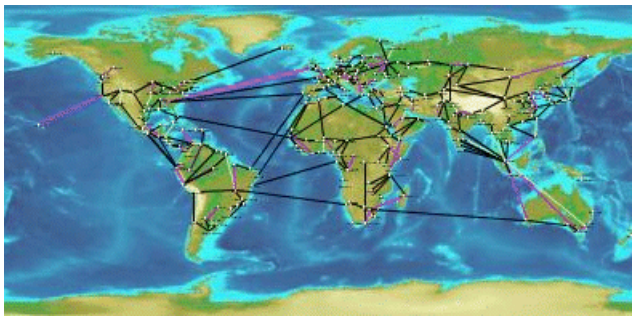
The getMaxZoomLevel Method

This method allows you to fix the user's maximum zoom level on the client side. Here we overwrite the method to return a larger value.

As you have seen, creating the servlet is very simple. This servlet can now answer HTTP requests from a client by sending JPEG images. If you have installed the example, you can try the following HTTP request:

```
http://localhost:8080/xmlgrapher/
demo.xmlgrapher.servlet.XmlGrapherServlet?request=image
            &format=JPEG&bbox=0,0,512,512
            &width=400
            &height=200
            &layer=Cities,Links,background%20Map
```

This produces the following image:



Generated Bitmap Image

This request asks the servlet named `demo.xmlgrapher.servlet.XmlGrapherServlet` to produce an image of size 400 x 200 showing the area (0, 0, 512, 512) of the manager with the layers "Cities," "Links," and "Background Map" visible.

In most cases, you do not have to know the servlet parameters because the Dynamic HTML objects or the Java™ classes provided by IBM® ILOG® JViews for the client side will take care of the HTTP requests for you.

This example is a very simple servlet. This servlet uses the same `IlvManagerView` instance for all clients; this means that every client will see the same data. For more complex usage of the `IlvManagerServlet` classes, read *The IlvManagerServlet class*.

Developing the client

Describes the client side of a thin-client application and how to develop a dynamic HTML client by adding JavaScript™ components.

In this section

Overview of client-side development
Describes how Dynamic HTML influences client-side development.

The IlvView JavaScript component
Describes the `IlvView` component.

The IlvOverview JavaScript component
Describes the `IlvOverview` component.

The IlvLegend JavaScript component
Describes the IlvLegend component.

The IlvButton JavaScript component
Describes the `IlvButton` component.

The IlvZoomTool JavaScript component
Describes the `IlvZoomTool` component.

The IlvZoomInteractor JavaScript component
Describes the `IlvZoomInteractor` component.

IlvPanInteractor
Describes the `IlvPanInteractor` component.

The IlvPanTool JavaScript component
Describes the `IlvPanTool` component.

The IlvMapInteractor and IlvMapRectInteractor JavaScript components
Describes the `IlvMapInteractor` and `IlvMapRectInteractor` components.

The Popup menu in JavaScript
Describes the JavaScript component for the popup menu.

Overview of client-side development

After creating the server (see *Developing the server*), you can create the client side. The IBM® ILOG® JViews thin-client support allows you to build a DHTML client easily. The static nature of HTML limits the interactivity of web pages. Dynamic HTML allows you to create more interactive and engaging web pages. It gives content providers new controls and allows them to manipulate the contents of HTML pages through scripting.

IBM® ILOG® JViews provides a set of Dynamic HTML components written in JavaScript™ that allows you to build your DHTML pages very easily. The JavaScript files are located in ***<installdir>*** **`/jviews-framework86/lib/thinclient/javascript`**.

Important: This sample is compatible with the browsers and browser versions listed in the Release notes under *Requirements for running thin-client applications*.

The Dynamic HTML client for the XML Grapher example includes most of the DHTML components. The full HTML file for the XML Grapher example is located in ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/index.html`**.

The full reference documentation of each component can be found in the *JavaScript Reference Manual* located in ***<installdir>*** **`/jviews-framework86/doc/html/en-US/refjsf/html/index.html`**.

The IlvView JavaScript component

The `IlvView` component (located in the `IlvView.js` file) is the main component. This component queries the servlet and displays the resulting image.

To use this component, you need to include the following JavaScript™ files: `IlvUtil.js` , `IlvView.js` , the files for the superclasses of `IlvView`: `IlvAbstractView.js`, `IlvResizableView.js`, and `IlvEmptyView.js`, and `IlvGlassView.js`.

Instead of including the individual `.js` files of each component, you can add the file `framework.js` which is located in *<installdir>* `/jviews-framework86/lib/thinclient/framework/framework.js`

This file is a concatenation of all the `.js` files required for doing DHML thin client in the Framework.

Here is a simple HTML page that creates an instance of `IlvView`:

HTML code

```
<html>
<head>
<META HTTP-EQUIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
</head>
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
<script TYPE="text/javascript">

function init() {
  view.init()
  return false
}

function handleResize() {
  if (document.layers)
   window.location.reload()
}
</script>
<body onload="init()" onunload=”IlvObject.callDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440
```

```
// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.toHTML()

</script>
</body>
</hmtl>
```

This example starts by importing some JavaScript files:

```
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
```

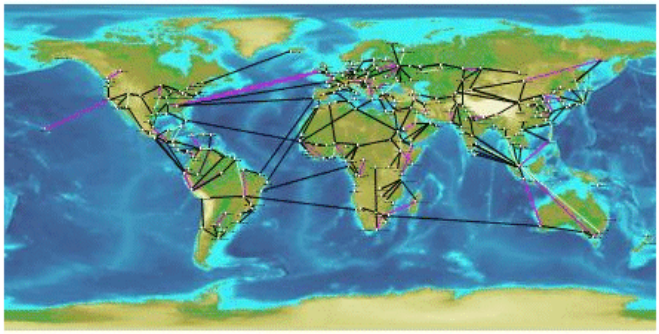
In the body of the page, the example creates an `IlvView` object located in (40, 40) on the HTML page. The size is 440 x 270. This view displays images produced by the servlet `XmlGrapherServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

This example also defines two JavaScript functions:

- The `init` function, called on the `onload` event of the page, initializes the `IlvView` by calling its `init` method.
- The `handleResize` function, called on the `onresize` event of the page, will reload the page if the browser is Netscape Communicator 4 or higher. This is necessary for a correct resizing of Dynamic HTML content on Communicator.

Note: The global `IlvObject.callDispose()` function must be called in the `onunload` event of the HTML page. This function disposes of all resources acquired by the JViews DHTML components.

Once the image is loaded from the server, the page now looks like this:



Generated HTML Page

The IlvOverview JavaScript component

The `IlvOverview` component (located in the `IlvOverview.js`) file shows an overview of the manager. An `IlvOverview` is linked to an `IlvView` component. By default, the `IlvOverview` queries the server to obtain an image of the global area and displays it. Once the overview is visible, a rectangle corresponding to the area visible in the main view is drawn on top of the overview. You can move this rectangle to change the area visible in the main view.

Here is the body of the previous example with an `IlvOverview` component. Note that you cannot move the rectangle of the overview now because the complete area is visible in the main view. You will be able to do that later when the zooming functionality is added.

Note: The lines added are in **bold**.

```
<body onload="init()" onunload=”IlvObject.callDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL(’/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet’)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor(’white’)

view.toHTML()
overview.toHTML()

</script>
```

Compared to the previous example, there is a new import statement for `IlvOverview.js`:

```
<script TYPE="text/javascript"  src="script/IlvOverview.js"></script>
```

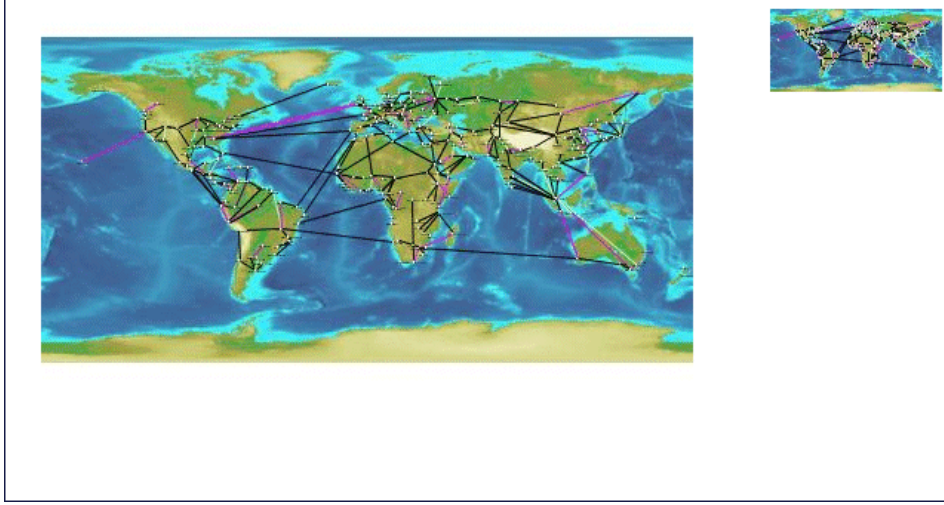
An `IlvOverview` object located in (x+w+50, y+4) with a size of 120 x 70 was created:

```
var overview = new IlvOverview(x+w+50, y+4, 120, 70, view)
```

The following line sets the color of the draggable rectangle:

```
overview.setColor('white')
```

The page looks now like this:



The IlvLegend JavaScript component

You can add an `IlvLegend` component to the page. The `IlvLegend` component shows a list of layers that are available on the server side, and allows you to turn the visibility of a layer on and off.

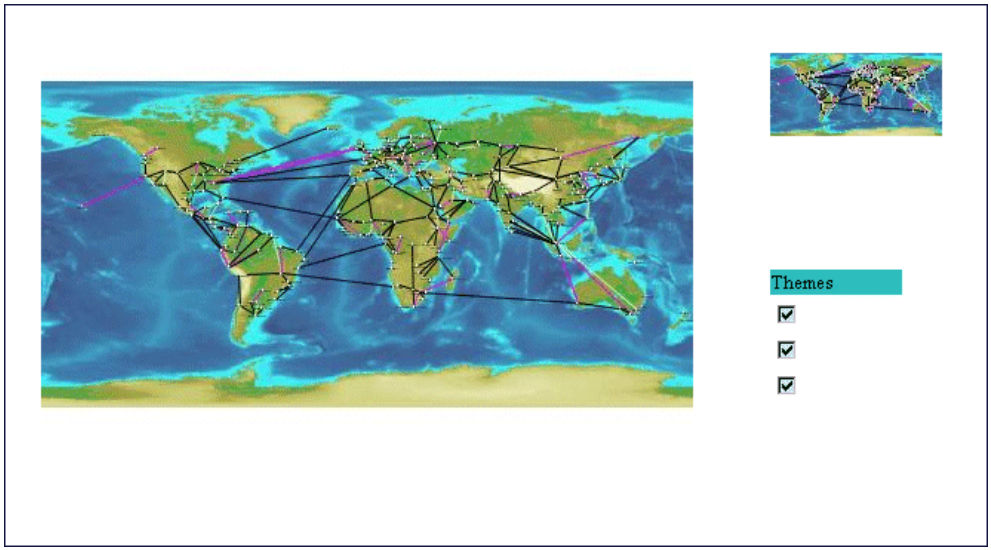
To use the `IlvLegend`, you must first include the `IlvLegend.js` file.

```
<script TYPE="text/javascript" src="IlvLegend.js"></script>
```

The body of the HTML file now looks like this:

```
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);

view.toHTML()
overview.toHTML()
legend.toHTML()
</script>
</body>
```

You should see the following page:



The visibility of layers can now be turned on and off.

The IlvButton JavaScript component

The `IlvButton` component is a simple button that allows you to call some JavaScript™ code by clicking it. You can add some buttons to the page to zoom in and out.

In addition to buttons, you can add some Dynamic HTML panels to create a frame around the main view. A Dynamic HTML panel is an area of the page that can contain some HTML. Creating a panel is done using the class `IlvHTMLPanel`, defined in the `IlvUtil.js` file.

The body of the page is now:

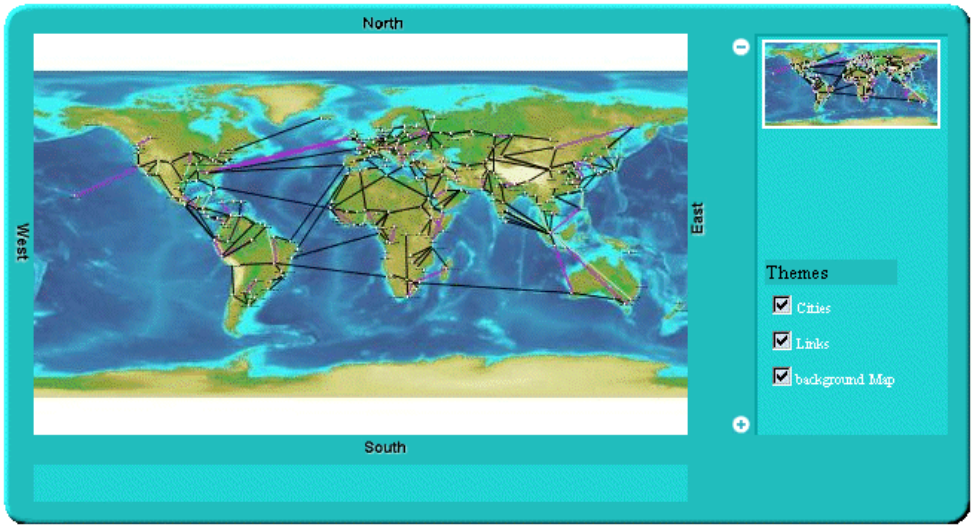
```
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff" >
<script>

//position of the main view

var y = 40
var x = 40
var h = 270
var w = 440

// Creates a frame around the main view
var frameBackground = new IlvHTMLPanel('')
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor('#21bdbd')

var frameTopLeft = new IlvHTMLPanel('<IMG src="images/frame_topleft.gif">')
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft =new IlvHTMLPanel('<IMG src="images/frame_bottomleft.gif">')
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel('<IMG src="images/frame_topright.gif">')
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel('<IMG src="images/frame_bottomright.
gif">')
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel('<IMG src="images/frame_top.gif">')
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel('<IMG src="images/frame_bottom.gif">')
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel('<IMG src="images/frame_left.gif">')
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)
var frameRight = new IlvHTMLPanel('<IMG src="images/frame_right.gif">')
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)

var border = new IlvHTMLPanel('')
border.setBounds(x+w+45, y, 130, h)
border.setVisible(true)
border.setBackgroundColor('#09a5a5')

var secondBorder = new IlvHTMLPanel('')
secondBorder.setBounds(x+w+47, y+2, 128, h-2)
secondBorder.setVisible(true)
secondBorder.setBackgroundColor('#21d6d6')

// message panel
var messagePanel = new IlvHTMLPanel('')
messagePanel.setBounds(x, y+h+20, w, 25)
messagePanel.setVisible(true)
messagePanel.setBackgroundColor('#21d6d6')
IlvButton.defaultMessagePanel = messagePanel;

// IBM® ILOG® logo
var logo = new IlvHTMLPanel('<IMG src="images/ilog.gif">')
logo.setBounds(x+w+95, y+h+10, 85, 40)
logo.setVisible(true)

IlvButton.defaultInfoPanel = messagePanel;

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.setMessagePanel(messagePanel)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')
overview.setMessagePanel(messagePanel)

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);
// Some buttons for navigation
var topbutton, bottombutton, rightbutton, leftbutton

topbutton = new IlvButton(x+w/2, y-15, 30, 13,'images/north.gif','view.panNorth
()')
```

```
topbutton.setRolloverImage('images/northh.gif')
topbutton.setToolTipText('pan north')
topbutton.setMessage('pan the map to the north')
```

```
bottombutton = new IlvButton(x+w/2, y+h, 33, 13,'images/south.gif','view.
panSouth()')
bottombutton.setRolloverImage('images/southh.gif')
bottombutton.setToolTipText('pan south')
bottombutton.setMessage('pan the map to the south')

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,'images/west.gif','view.panWest
()')
leftbutton.setRolloverImage('images/westh.gif')
leftbutton.setToolTipText('pan west')
leftbutton.setMessage('pan the map to the west')

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, 'images/east.gif', 'view.
panEast()')
rightbutton.setRolloverImage('images/easth.gif')
rightbutton.setToolTipText('pan east')
rightbutton.setMessage('pan the map to the east')

// Buttons to zoom in and out
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, 'images/zoom.gif', 'view.
zoomIn()')
zoominbutton.setRolloverImage('images/zoomh.gif')
zoominbutton.setMessage('click to zoom by 2')
zoominbutton.setToolTipText('Zoom In')

zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif', 'view.
zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')

view.toHTML()
overview.toHTML()
legend.toHTML()
topbutton.toHTML()
bottombutton.toHTML()
leftbutton.toHTML()
rightbutton.toHTML()
zoomoutbutton.toHTML()
zoominbutton.toHTML()

</script>
</body>
</hmtl>
```

The page now looks like this:



A frame around the page was created by the following lines:

```
var frameBackground = new IlvHTMLPanel('')
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor('#21bdbd')

var frameTopLeft = new IlvHTMLPanel('<IMG src="images/frame_topleft.gif">')
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft=new IlvHTMLPanel('<IMG src="images/frame_bottomleft.gif">')
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel('<IMG src="images/frame_topright.gif">')
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel('<IMG src="images/frame_bottomright.
gif">')
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel('<IMG src="images/frame_top.gif">')
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel('<IMG src="images/frame_bottom.gif">')
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel('<IMG src="images/frame_left.gif">')
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)

var frameRight = new IlvHTMLPanel('<IMG src="images/frame_right.gif">')
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)
```

This creates four DHTML panels for the corners, four additional panels for the sides, and a panel for the background. The corners and the sides of the frame are composed of simple GIF images.

Four buttons to pan south, north, east, and west have been added by the lines:

```
topbutton = new IlvButton(x+w/2, y-15, 30, 13,'images/north.gif','view.panNorth
()')
topbutton.setRolloverImage('images/northh.gif')
topbutton.setToolTipText('pan north')
topbutton.setMessage('pan the map to the north')

bottombutton = new IlvButton(x+w/2, y+h, 33, 13,'images/south.gif','view.
panSouth()')
bottombutton.setRolloverImage('images/southh.gif')
bottombutton.setToolTipText('pan south')
bottombutton.setMessage('pan the map to the south')

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,'images/west.gif','view.panWest
()')
leftbutton.setRolloverImage('images/westh.gif')
leftbutton.setToolTipText('pan west')
leftbutton.setMessage('pan the map to the west')

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, 'images/east.gif', 'view.
panEast()')
rightbutton.setRolloverImage('images/easth.gif')
rightbutton.setToolTipText('pan east')
rightbutton.setMessage('pan the map to the east')
```

A button is defined by its position and size, two images, the main image and the rollover image, and a piece of JavaScript to be executed when the button is clicked.

Note that in order to pan to the north, you use the `panNorth` method of `IlvView`.

Two additional buttons have been created to zoom in and out, by the lines:

```
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, 'images/zoom.gif', 'view.
zoomIn()')
zoominbutton.setRolloverImage('images/zoomh.gif')
zoominbutton.setMessage('click to zoom by 2')
zoominbutton.setToolTipText('Zoom In')
```

```
zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif', 'view.
zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')
```

Each button has a `message` property. The message will be automatically displayed in the status window of the browser when the mouse is over the button. The message can also be displayed in an additional panel. This is why the line:

```
IlvButton.defaultInfoPanel=messagePanel
```

tells you that messages of buttons will also be displayed in the DHTML message panel.

The IlvZoomTool JavaScript component

The `IlvZoomTool` component is a DHTML 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, and the images of the buttons can be customized.

To add the component, add the following lines to the page:

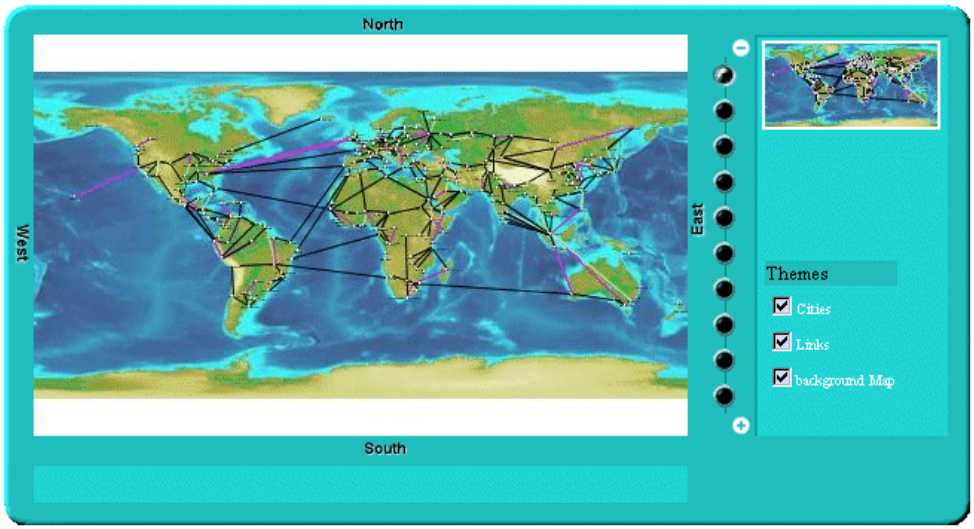
```
<script TYPE="text/javascript" src="script/IlvZoomTool.js"></script>
```

This line imports the script.

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

```
var zoomtool = new IlvZoomTool(x+w+25, y+15, 25, h-30, 10 , view)
zoomtool.setOrientation('Vertical')
zoomtool.upImage = 'images/button.gif'
zoomtool.rolloverUpImage = 'images/buttonh.gif'
zoomtool.downImage = 'images/button.gif'
zoomtool.rolloverDownImage = 'images/buttonh.gif'
zoomtool.currentImage = 'images/center.gif'
zoomtool.rolloverCurrentImage = 'images/centerh.gif'

zommtool.toHTML()
```

The page now looks like this, with the vertical zoom tool on the right of the main view:



The IlvZoomInteractor JavaScript component

The `IlvZoomInteractor` allows direct interaction with the image; it allows the user to select an area on the image to zoom this area. Installing an interactor on the view is simple: you need only create the interactor and set it to the view:

```
var zoomInteractor = new IlvZoomInteractor()
view.setInteractor(zoomInteractor)
```

In the example, you add a button that will install the interactor. To do this, add the following lines to the page:

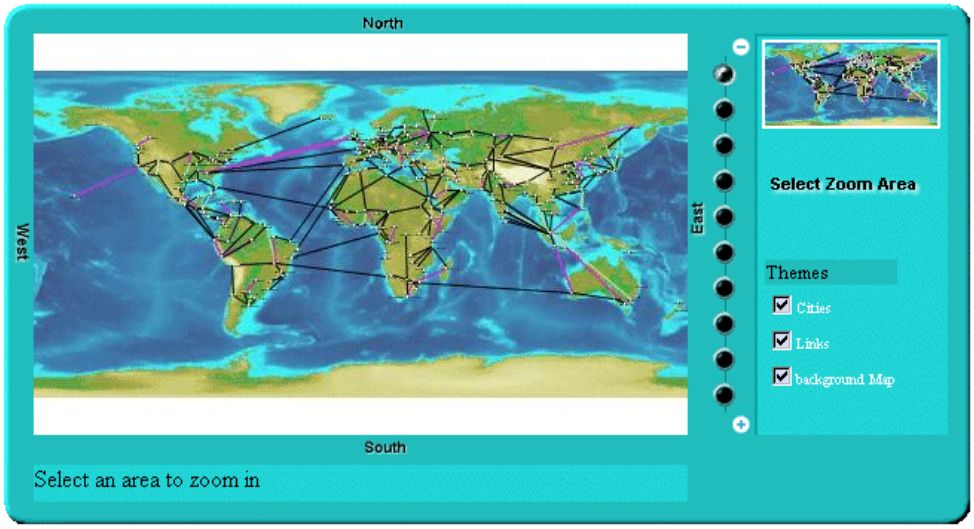
```
<script TYPE="text/javascript"
      src="script/IlvInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvDragRectangleInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvZoomInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvInteractorButton.js"></script>
```

To use the interactor, you have to import three JavaScript™ files: `IlvInteractor.js`, `IlvDragRectangleInteractor.js`, and `IlvZoomInteractor.js`. This is because the `IlvZoomInteractor` component is a subclass of the `IlvDragRectangleInteractor` component.

Then you add the following lines to the body of the page:

```
var zoomInteractor = new IlvZoomInteractor()
zoomInteractor.setLineWidth(1)
zoomInteractor.setColor('#00ffff')

...

var zoomrectbutton

zoomrectbutton=new IlvInteractorButton(x+w+50, y+90, 112, 24,
                                       'images/zoomrect.gif', zoomInteractor,
view)
zoomrectbutton.setRolloverImage('images/zoomrecth.gif')
zoomrectbutton.setMessage('click to set zoom mode')
zoomrectbutton.setToolTipText('Zoom Mode')

...

zoomrectbutton.toHTML()
```

This results in the following page:



You can now click the “Select Zoom Area” button to install the interactor and then select an area to zoom in.

IlvPanInteractor

The `IlvPanInteractor` component allows the user to click in the main view to pan the view. Just as for the `IlvZoomInteractor`, use the `setInteractor` method of `IlvView` to install the interactor. In the example, add another button that will install this interactor (see *The IlvZoomInteractor JavaScript component*). You will now be able to switch from the "Pan" mode and the "Zoom" mode.

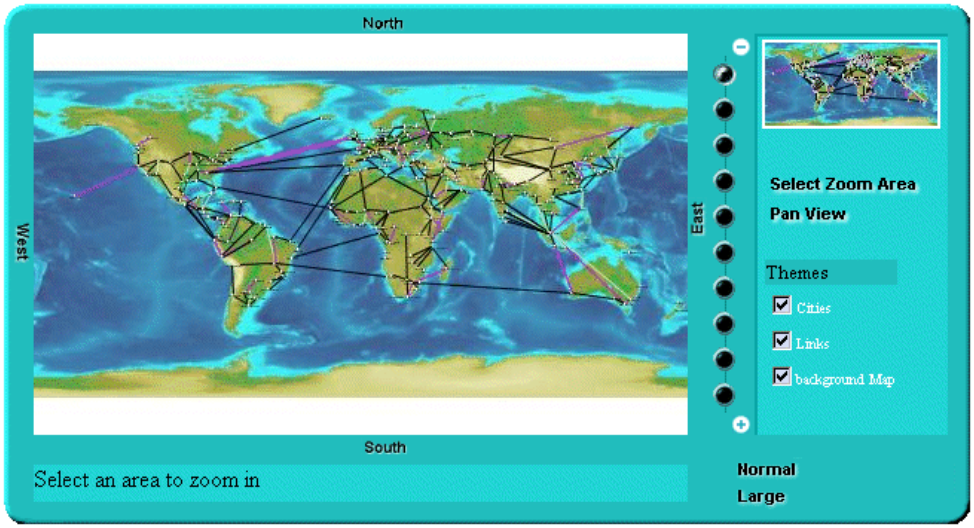
To be able to use the component, import the corresponding JavaScript™ file:

```
<script TYPE="text/javascript"
      src="script/IlvPanInteractor.js"></script>
```

Then add the following lines to the body of the page:

```
var panInteractor  = new IlvPanInteractor()
panbutton=new IlvInteractorButton(x+w+50, y+110, 63, 22, 'images/pan.gif',
                                  panInteractor, view)
panbutton.setRolloverImage('images/panh.gif')
panbutton.setMessage('click to set pan mode')
panbutton.setToolTipText('Pan Mode')
...

panbutton.toHTML()
```

The page now has one additional button labelled "Pan View":



The example is now complete; it uses most of the DHTML components provided by IBM® ILOG® JViews.

The IlvPanTool JavaScript component

The `IlvPanTool` component (located in the `IlvPanTool.js` file) is a component that allows panning of the view in all directions. You create the component in this way:

```
var pantool = new IlvPanTool(10, 10, view)
pantool.toHTML()
```

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

This component looks like this:

The IlvMapInteractor and IlvMapRectInteractor JavaScript components

The `IlvMapInteractor` and `IlvMapRectInteractor` components are two additional interactors that can be used to perform an action on the server side when a point or an area of the image is selected by the client. These interactors and how to use them are described in detail in *Adding client/server interactions*.

The Popup menu in JavaScript

The popup menu component is attached to the main view. This popup menu in JavaScript™ is triggered by a right-click in the view.

To use the popup menu, you must first include the following scripts.

The popup menu can be contextual or static.

Static popup menu

The menu is static, that is, not conditioned by the context in which it is called, and is defined in the HTML file by using `IlvMenu` and `IlvMenuItem` instances. The menu is a pure client-side object and there is no roundtrip to the server to generate the menu.

Contextual popup menu

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

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

On the client side, you need only declare the popup menu and set it on the view.

The factory must implement the `IlvMenuFactory` interface.

Styling the popup menu

You can style the popup menu by setting a CSS class name in the following properties:

- `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 example shows how to use CSS to style the popup menu.

Adding client/server interactions

Overview of actions on the server and client sides

The IBM® ILOG® JViews thin-client support gives you a simplified way to define new actions that should take place on the server side. For example, suppose you want to allow the user to delete a graphic object that appears on the generated image. Part of this action—clicking the image to select the object—must be done on the client side. The destruction of the object must be done on the server side before a new image is generated. The notion of "server-side action" exists to perform such behavior. An action is defined by a name and a set of string parameters.

Actions on the client side

In a dynamic HTML client, you tell the server to perform an action using the `performAction` method of the `IlvView` JavaScript™ component.

Here is an example that asks the server side to execute the action "delete" with coordinate parameters, assuming that `view` is an `IlvView`:

```
var x = 100;
var y = 50;
var params = new Array();
params[0]=x;
params[1]=y;
view.performAction("delete", params);
In a thin-Java client the system is the same:
float x = 100f;
float y = 50f;
String[] params = new String[2];
params[0] = Float.toString(x);
params[1] = Float.toString(y);
view.performAction("delete", params);
```

The `performAction` method will ask the server for a new image. In the image request, additional parameters are added so that the server side can execute the action. Thus, the `performAction` call results in only one client/server round-trip.

Note that predefined interactors are provided to help you define new actions on the client side. They are explained in *Predefined interactors*.

Actions on the server side

On the server side, you need to detect that an action was requested and execute the action. This is done using the interface `ServerActionListener`.

To be able to listen and execute an action on the server side, you simply add an action listener to your servlet. In the `performAction` method of the listener, you check the action name and perform the action.

For the “delete” action, we would add the following lines of code in the `init` method of the servlet:

```
addServerActionListener(new ServerActionListener() {
   public void actionPerformed(ServerActionEvent e) throws ServletException
  {
    if (e.getActionName().equals("delete")) {
      IlvPoint p = e.getPointParameter(0);
        // find object under this point and delete it if there is one.
    }
  }
});
```

The `ServerActionEvent` object can give you all necessary information about the action, the name, and its parameters.

Predefined interactors

Two predefined interactors are provided to help you create new actions: `IlvMapInteractor` and `IlvMapRectInteractor`.

`IlvMapInteractor` allows the user to click in the map; it will ask the server to execute an action, with the coordinates of the clicked point passed as parameters. The second interactor is almost the same except that the user selects an area of the image instead of clicking on it.

Generating a client-side image map

If you are creating a Dynamic HTML client, the IBM® ILOG® JViews thin-client support allows you to create a client-side image map. Image maps are images with an attached map that points out hot spots, or clickable areas. In the IBM® ILOG® JViews thin-client support, a clickable area can be generated for each graphic object of the manager.

To create a client side image map:

- Define the image map on the server side
- Use the image map on the client side

Define the image map on the server side

The servlet provided by IBM® ILOG® JViews (`IlvManagerServlet`) is able to generate an image map for your IBM® ILOG® JViews application, but it is likely that you do not want to generate a clickable area for every graphic object. On the server side, you will then have to tell the manager servlet which IBM® ILOG® JViews layer and which graphic object are part of the image map generation. For both layer and graphic object, this is done by setting a property on them.

On a layer, assuming that the variable `manager` is an `IlvManager`, you will do:

```
manager.getManagerLayer(index).setProperty( IlvManagerServlet.
ImageMapAreaGeneratorProperty, Boolean.TRUE);
```

On a graphic object you can do almost the same thing, but the value of the property must be an instance of the class `IlvImageMapAreaGenerator`. This class is responsible for generating the AREA part of the image map.

Note that the same instance of `IlvImageMapAreaGenerator` can be used for all graphic objects.

By default, `IlvImageMapAreaGenerator` will generate a rectangular area with no HREF in it. You will have to subclass it to generate an HREF for your graphic object.

Here is an example that creates a custom `IlvImageMapAreaGenerator` and sets it on some objects:

```
IlvGraphic object1, object2;
....
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {
    public String generateHREF(IlvManagerView v, IlvGraphic obj) {
    String href;
    // place here code the
    // computes the URL depending on the graphic object
    return href;
  }

};
```

```
object1.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                    generator);
object2.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                    generator);
```

The HREF can be a URL to which the browser will jump when the area is clicked, but it can also be a call to a JavaScript™ method.

For example, in the XML Grapher example, you can define the generator like this:

```
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {

  public String generateALT(IlvManagerView v, IlvGraphic obj) {
     return ((GrapherNode)obj).getLabel();
  }

  public String generateHREF(IlvManagerView v, IlvGraphic obj) {
     return "javascript:doSomething('"+
             ((GrapherNode)obj).getLabel()+"')";
  }
};
```

In this example, the HREF generated is a call to the JavaScript method `doSomething`. You will have to define this method in the HTML page.

For more information about customizing an area, see the `IlvImageMapAreaGenerator` class in the *Java API Reference Manual*.

Use the image map on the client side

To tell the Dynamic HTML client to generate a client-side image map, you only need to set the `imageMap` property of the `IlvView` JavaScript™ component to `true`:

```
var view = new IlvView(40, 40, 300, 400);
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet');
view.setGenerateImageMap(true);
```

When this is done, the `IlvView` component will ask the servlet to generate the image map.

To make the image map visible, there are two possibilities. You can:

- Directly call the `showImageMap` method of `IlvView`:

  ```
  view.showImageMap();
  ```

- Use the `IlvImageMapInteractor` class. This class is a simple interactor that will show the image map when installed and hide it when de-installed.

The IlvManagerServlet class

Describes the predefined servlet and how to use it.

In this section

Overview of the predefined servlet
Presents the predefined servlet.

The servlet requests and parameters
Presents the requests to which the servlet can respond and the parameters they take.

Multiple sessions
Describes the need for multiple sessions and gives an example.

Multithreading issues
Describes the use of single-thread and multithread versions of servlets and resulting synchronization requirements.

Overview of the predefined servlet

Developing the server side of a thin-client application consists of creating a servlet that can produce an image to the client. IBM® ILOG® JViews Framework provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

The `IlvManagerServlet` class is an abstract Java™ subclass of the `HTTPServlet` class from the Java servlet API.

The servlet requests and parameters

The servlet can respond to three different types of HTTP requests, the “image” request, the “image map” request, and the “capabilities” request. The image request will return an image from the IBM® ILOG® JViews manager. The capabilities request will return information to the client, such as the layers available in the manager and the global area of the manager. This information allows the client to know the capabilities of the servlet in order to build the image request. When developing the client side of your application, you will use the DHTML scripts or the JavaBeans™ provided by IBM® ILOG® JViews; both will create the HTTP request for you, so you do not really need to write the HTTP request yourself.

The image request

The image request produces a JPEG image from the manager. The request has the following syntax, assuming that `myservlet` is the name of the servlet:

```
http://host/myservlet?request=image
   &bbox=x,y,width,height (area in the manager coordinate system)
   &width=width of the returned image
   &height=height of the returned image
   &layer=comma separated list of layers
   &format=JPEG
   &bgcolor=0xFFFFFF
```

Here is a list of parameters and their meanings.

Parameters of the IlvManagerServlet

Parameter Name	Parameter Value	Description
`request`	image	Asks the servlet to generate an image.
`bbox`	Float, Float, Float, Float	The area of the manager that will be displayed in the image. The first two values are the upper left

Parameter Name	Parameter Value	Description
		corner of the area. The last two values are the width and height of the area.
`width`	Integer	Width of the resulting image.
`height`	Integer	Height of the resulting image.
`format`	JPEG	The format of the resulting image.
`layer`	Comma-separated list of strings. For example: Cities, Roads	The layers of the `IlvManager` that will be visible.
`bgcolor`	`0xrrggbb` For example, `0xffffff` for white	The background color of the resulting image. This parameter is optional.
`action`	actionName(param1, param2)	Specifies an action to be executed on the server before the image is generated.

The following request will produce a JPEG image of size (250, 250) showing the area (0, 0, 1000, 1000) of the manager; only the layers named “Cities” and “Roads” will be visible:

```
http://host/myservlet?request=image
        &bbox=0,0,1000,1000
        &width=250
        &height=250
        &layer=Cities,Roads
        &format=JPEG
```

The capabilities request

The capabilities request produces information to the client. This request returns information on the manager.

The capabilities request has the following syntax:

```
http://host/myservlet?request=capabilities
        &format=(html|octet-stream)
      [ &onload= <a string> ]
```

The `request` parameter set to `capabilities` instead of `image` tells the servlet to return the capabilities information. The `format` parameter tells which format should be returned.

The result can be of two different formats, HTML or Octet stream.

HTML format

The HTML format is used when the client is a Dynamic HTML client. In this case, the result is a empty HTML page that contains some JavaScript™ code. The JavaScript code is executed on the client side, and some information variables are then available.

```
<html>
<head>
<script language="JavaScript">
var minx=0.0;
var miny=0.0;
var maxx=1024.0;
var maxy=512.0;
var themes=new Array();
var overviewthemes=new Array();
themes[0]="a layer name";
overviewthemes[0]=true;
themes[1]="another layer";
overviewthemes[1]=true;
themes[2]="a third layer";
overviewthemes[2]=true;
var maxZoom=6;
</script>
</head>
<body>
</body>
</html>
```

The variables `minx`, `miny`, `maxx`, `maxy` are defining the global area of the manager that can be queried. The `themes` variable is the list of layers available on the server side. The `overviewthemes` variable tells if a layer should be visible in the overview window. The `maxZoom` variable is the maximum level of zoom the application should perform.

The `onload` parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an `onload` parameter is specified, the body tag of the HTML page is the following:

```
<body onLoad="+onload+">
```

Octet-stream format

The octet-stream format is used when the client is a Java™ applet. In this case, the result is a stream of octets. The data is produced using a `java.io.DataOutput` and can be read using a `java.io.DataInput`. It is organized as follows:

```
Float: left coordinate of manager's bounding box.
   Float: top coordinate of manager's bounding box.
   Float: right coordinate of manager's bounding box.
   Float: bottom coordinate of manager's bounding box.
   Int: number of layers.

   for each layer:
```

```
       String (UTF format): name of the layer.
       Boolean: is the layer an overview layer.

    Float: Maximum zoom level
```

You see that this format gives the same type of information as the HTML format. Once again, you do not need to decode or read these formats. The client-side components provided by IBM® ILOG® JViews will do that for you.

The image map request

The image map request produces an image and a client-side image map. The parameters for this request are the same as for the image request except that the `request` parameter must have the value `imagemap`.

For example, the following code to the servlet:

```
http://host/myservlet?request=imagemap
        &width=400
        &height=200
        &bbox=0,0,500,500
        &format=JPEG
        &layer=Cities,Links,background%20Map
```

will produce something like:

```
<html>
<body>
<map name="imagemap">
<area shape="rect" coords="242,81,261,83" href="..." >
....
</map>
<img usemap="#imagemap" width="400" height="200"
  src="myservlet?request=image&layer=Cities,Links,background%20Map&width=400
  &format=JPEG&bbox=0,0,500,500&height=200" border=0>
</body>
</html>
```

The call generates an HTML document containing the client-side image map and an image. The contents of the image are then generated by another call to the servlet.

The graphic objects that are taken into account when generating the map can be specified as well as the shape of the clickable area and what appends when you click on it. All this is explained in *Generating a client-side image map*.

The image map request has two additional optional parameters:

- The `mapname` parameter allows you to specify the name of the map. The default name is `imagemap`.
- The `onload` parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an `onload` parameter is specified, the body tag of the HTML page is the following:

```
<body onLoad="+onload+">
```

Multiple sessions

The XML Grapher is a very simple example that creates a single manager view for the servlet. This means that all calls to the servlet (that is, all clients) are looking at the same view. This is fine when the same data is used for all clients but in some applications—for example, when you want to allow the user to edit the graphic representation—you might want to have a view (and thus a manager) for each client. In this case, you might use the notion of *HTTP sessions*. You can then create a view and a manager and store them as parameters of the session.

Here is a slightly modified version of the XML Grapher servlet using sessions:

```
package demo.xmlgrapher.servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import ilog.views.*;
import ilog.views.servlet.*;
import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  String xmlfile;

  public void init(ServletConfig config)
       throws ServletException
  {

    xmlfile = config.getInitParameter("xmlfile");
    if (xmlfile == null)
      xmlfile = config.getServletContext().
                    getRealPath("/data/world.xml");
    setVerbose(true);
  }

  protected void prepareSession(HttpServletRequest request)
  {
    HttpSession session = request.getSession();
    if (session.isNew()) {

      XmlGrapher xmlGrapher = new XmlGrapher();
      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }
      session.putValue("IlvManagerView", xmlGrapher);
    }
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
```

```
    HttpSession session = request.getSession(false);
    if (session!= null)
      return (IlvManagerView)session.getValue("IlvManagerView");
    else
      throw new ServletException("session problem");
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                  IlvManagerView view)
  {
    return 30;
  }
}
```

The `init` method does not create any `XmlGrapher` object any more. Instead, the `prepareSession` method (which has a default empty implementation) is overwritten to get the HTTP session. If this is a new session, an `XmlGrapher` object is created and stored as a parameter of the session. The `getManagerView` method returns the `XmlGrapher` object stored in the session.

Multithreading issues

The `IlvManagerServlet` class does not implement the `SingleThreadModel` interface from the Servlet API, so you can create servlets that use the multithread or single-thread model.

If your servlet implements the `SingleThreadModel` interface, then you do not have to deal with concurrent access to your servlet. The servlet will be thread safe. However, this interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

If your servlet does not implement the `SingleThreadModel` interface, then you might have to be concerned with concurrent access to the servlet. All basic operations done by the `IlvManagerServlet` on the `IlvManagerView` are already synchronized. This means that you will have to take care of concurrent access only if you are doing additional actions on the `IlvManagerView`. In this case you can define a locking object and use the `getLock` method of the `IlvManagerServlet`. Each request handling is implemented in the following way:

```
... reads the request parameters ...

synchronized(getLock(request)) {
  IlvManagerView view = getManagerView(request);

  ... handle the request ...
}
```

By default, the `getLock` method returns a new object each time. This means that the section is not synchronized.

The IlvManagerServletSupport class

The `IlvManagerServlet` class used in the XML Grapher example gives an easy way to create a servlet that supports the IBM® ILOG® JViews thin-client protocol. Using the `IlvManagerServlet` class is an easy way to create a servlet but has one main drawback. You cannot add the support for the IBM® ILOG® JViews thin-client protocol to an existing servlet since the `IlvManagerServlet` class derives from the `HttpServlet` class. The `IlvManagerServletSupport` class will allow you to do this. This class has the same API as the `IlvManagerServlet` but is not a servlet (that is, it does not derive from the `HttpServlet` class). You can thus create your own servlet and an instance of the `IlvManagerServlet` support class in this servlet to handle the requests coming from the IBM® ILOG® JViews client side.

Thin-client support in the XML Grapher example

In the XML Grapher example, the code of the servlet can be rewritten using the `IlvManagerServletSupport` class as follows:

```
package demo.xmlgrapher.servlet;

import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;
import java.io.*;
import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends HttpServlet
{
  IlvManagerServletSupport servletSupport ;

  class MySupport extends IlvManagerServletSupport {

    private XmlGrapher xmlGrapher;

    public MySupport(ServletConfig config) {
      super();
      xmlGrapher = new XmlGrapher();

      String xmlfile = config.getInitParameter("xmlfile");

      if (xmlfile == null)
        xmlfile = config.getServletContext().getRealPath("/data/world.xml");

      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }
```

```
      setVerbose(true);
    }

    public IlvManagerView getManagerView(HttpServletRequest request)
      throws ServletException {
        return xmlGrapher;
      }

    protected float getMaxZoomLevel(HttpServletRequest request,
                                    IlvManagerView view) {
      return 30;
    }
  }

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException {
    servletSupport = new MySupport(config);
  }
```

```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    if (!servletSupport.handleRequest(request, response))
      throw new ServletException("unknow request type");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
       throws IOException, ServletException {
    doGet(request, response);
  }

}
```

This code creates a new servlet class, `XmlGrapherServlet`, that derives directly from the `HttpServlet` class. The `doGet` method passes the requests to an instance of the `IlvManagerServletSupport` class.

Specifying fixed zoom levels on the client side

Override the following method of the `IlvManagerServletSupport` class to specify the zoom levels that must be used on the client side:

```
public double[] getZoomLevels(HttpServletRequest request, IlvManagerView view)
```

In this case, the maximum zoom level is not used.

Controlling tiling

Describes how to control tiling on the client side and the server side.

In this section

Tiling
Explains what tiling is and its advantages.

Tile size
Explains tile size and its implications for performance and caching.

Cache mechanisms
Explains the cache mechanisms you can apply.

Developing client-side tiling
Describes how to develop the code on the client side if you use tiling.

Developing server-side tiling
Describes how to develop the code on the server side if you use tiling.

Client-side caching
Describes how to develop code for caching on the client side by managing HTTP headers.

Server-side caching and the tile manager
Describes how to develop code for caching on the server side by using a tile manager.

Tiling

The static layers are represented by a grid of images of a fixed size. These fixed-size images are referred to as tiles. Dynamic layers are represented by a single image with a transparent background overlaying the view.

A static layer is not supposed to change during the application lifecycle and so can be generated once only. Typically, a static layer is the background of the view, such as a background map.

A dynamic layer contains objects, such as symbols, that can move and change their graphic representation.

> **Note**: Dynamic layers must be placed on top of a static layer. Otherwise, they are not displayed.

The advantages of a tiled view are continuous panning and the capability of caching tiles. On the client side this avoids a roundtrip to the server and gives a better response time. On the server side it allows the server to receive the request, retrieve the image, and respond with the image without having to generate it. Not having to generate the image for the response is especially advantageous in complex applications.

Tile size

The size of the tile determines the number of tiles needed to cover the view.

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.

Cache mechanisms

Since tiles in static layers are not subject to change, they can be cached on the client side to be reused directly without the need for a server roundtrip.

You can consider several possible caching strategies on the server side:

- No caching: the server generates the images each time they are requested.
- Dynamic caching: the server can cache every generated tile, for example in the file system. This strategy allows you to have a quicker response for popular tiles and to limit the size of the cache.
- Pregeneration: a partial or complete set of tiles for specific zoom levels can be pregenerated and returned directly by the server without need of dynamic generation.

To manage the cache efficiently on the client and the server, the zoom levels must be fixed. If there is a free choice of what zoom level to apply, the probability of the client retrieving a cached tile is severely limited.

See *Specifying fixed zoom levels on the client side* for how to specify the zoom levels.

Developing client-side tiling

The API of the `IlvTileView` class is very similar to `IlvView`. To use the tiled view, import `IlvTiledView.js` instead of `IlvView.js`.

To instantiate an `IlvTiledView` object, proceed as with `IlvView`, but the class takes an additional argument that defines the tile size as shown in the following XML example.

```
<html>
<head>
<META HTTP-EQIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQIV="pRAGMA" CONTENT="No-cache">
</head>
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript" src="script/IlvTiledView.js"></script>
<script TYPE="text/javascript">
function init() {
  view.init()
  return false
}

function handleResize() {
  if (document.layers)
    window.location.reload()
}
</script>
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

//tile size
var t = 256

//Main view
var view = new IlvView(x,y,w,h,t)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.toHTML()
</script>

</body>
</html>
```

Developing server-side tiling

The tile manager stores and retrieves static and dynamic layers. See In *Server-side caching and the tile manager* for a description of the tile manager and Tiling for what is meant by static and dynamic layers in the context of tiling.

The list of dynamic layers is computed by the following method of the `IlvManagerServletSupport` class:

```
public IlvManagerLayer[] getDynamicLayers(HttpServletRequest request,
                 IlvManagerView view)
```

The default implementation of this method classifies the layers according to the value returned by the `getTripleBufferedLayerCount()` method. If the layer index is greater or equal to this value, the layer is dynamic. If not, it is a static layer. You can override this method to determine which are the dynamic layers in a different way.

Client-side caching

HTTP headers are sent with the tile image to control the caching of tiles on the client side.

There are two ways of specifying expiry data for tiles on the client side.

- Override the following method of `IlvManagerServletSupport`:

  ```
  public long getExpirationDate(HttpServletRequest request)
  ```

  This method returns the expiry date in milliseconds of tile lifespan in the client-side cache.

- Override the protected method:

  ```
  void setImageResponseCachePolicy(HttpServletRequest request,
  HttpServletResponse response);
  ```

  This method sends the HTTP headers to the client, so that the server instructs the client how to cache the tiles.

See RFC 2616 on HTTP/1.1 for a full description of HTTP headers.

You need to take the following cases into account:

1. The normal image request: you should prevent caching in this case.
2. The tile image request, which is identified by the `tile` request parameter: this type of request can be cached on the client.

Server-side caching and the tile manager

Use `IlvTileManager` to manage caching on the server side.

Static or dynamic layers can be used in conjunction with tiled views on the client side.

Static layers can be cached or pregenerated on the server. Cached tiles are part of layers that are not expected to change within the application lifecycle, as, for example, in a background map. Cached tiles can be retrieved through a tile manager.

Dynamic layers are likely to change between requests to the server, such as labeling or network display.

The tile manager, an instance of `IlvTileManager`, stores and retrieves tiles on the server side. `IlvManagerServlet` can take advantage of such a tile manager if one is installed on the servlet.

When an image request is received by the servlet, if a tile that matches the current request is managed by the tile manager, it will return this cached tile instead of generating a new image from `IlvManagerView`. If a tile is not yet managed by the tile manager, generate the image from `IlvManagerView` and ask the tile manager to manage it for future access.

When `IlvManagerServletSupport` responds to an image request, it uses the tile manager as follows:

```
if (useTileManager(request)) {
   IlvTileManager tm = getTileManager(request);
   if (tm != null) {
    Object key = getKey(request);
    BufferedImage image = tm.getImage(key);
    if (image == null) {
      image = doGenerateImageImpl( ... );
      tm.putImage(key, image);
     }
    return image;
   }
  }
  return doGenerateImageImpl( ... );
```

The tile manager is invoked by default if the request contains a parameter of the form `tile=true`. If the request contains such a parameter, `useTileManager(javax.servlet.http.HttpServletRequest)` will return `true`. You can override the `useTileManager` method to call the tile manager in other situations.

If a tile manager is installed, it will be retrieved and a key object will be constructed from the request to reference the tile. Then, an attempt is made to retrieve a tile from the tile manager. If the attempt is successful, the tile is returned as the response to the request.

If no tile is retrieved, an image will be constructed through the normal image generation process. This image is passed to the tile manager for use in future retrievals.

The tile manager is not installed by default in an `IlvManagerServletSupport` object. You need to subclass it to install a tile manager.

The method to override is `getTileManager(javax.servlet.http.HttpServletRequest)`. By default, this method returns null.

```
protected IlvTileManager getTileManager(HttpServletRequest request)
                throws ServletException {
   return null;
}
```

A default implementation of the tile manager is supplied. This implementation stores tiles on disk. You can use it to develop your own implementation of the `getTileManager` method.

```
protected IlvTileManager getTileManager(HttpServletRequest request)
                throws ServletException {
   ServletContext context = request.getSession().getServletContext();
   IlvTileManager tileManager =          (IlvTileManager)context.getAttribute
("tileManagerKey");
   if(tileManager == null) {
     tileManager = new IlvFileTileManager(getBase(), getMaxCacheSize(),
      getMinCacheSize());
     context.setAttribute("tileManagerKey", tileManager);
   }
   return tileManager;
}
```

In this implementation you need to provide:

- The base directory where the tiles are written.
- The maximum size allowed for the cache.
- The size to which the cache will be reduced by removing files when the maximum size is reached.

  When the maximum size is reached, the cache is considered to be full and files will be removed to reduce the size of the cache to the level indicated.

The tile manager is stored and retrieved from the `ServletContext`, so that the same tile manager is used for the same application. You can use a different strategy for storing and retrieving the tile manager.

You can also customize the reading and writing of tiles and the name of the file that is generated for each tile. This default implementation of the tile manager constructs a file name of the form `x_y_width_height.jpg`, where `x`, `y`, `width`, and `height` are the manager coordinates of the image request passed as the `bbox` attribute of the request.

This file is stored in and retrieved from the base directory provided when the `IlvFileTileManager` is constructed. This customization can be performed through the `IlvFileTileURLFactory`, which is responsible for building a URL from the key that identifies the tile. The default key is a `Rectangle2D.Double` object, which is created from the `bbox` parameter of the request.

Creating Rich Web Charts

The Rich Web Charts component is made of Rich Web Client and Server components that are used to display charts and interact with them.

In this section

Rich Web Charts
Describes the Rich Web Charts component.

Architecture overview
Provides an overview of the architecture of a Rich Web Charts application.

Getting started
This section provides a walkthrough of the basic steps required to create and deploy a Rich Web Charts application.

How to...
This section shows you how to use more of the features of the Rich Web Charts application.

The tag library
Describes the JSP tag library of the Rich Webs Charts component.

Client-side API
In response to user actions on the client-side, either on the chart component or any other JSF component, it is possible to use some JavaScript code to perform operations. For that purpose the Rich Web Charts client provides some JavaScript objects that are available in the scripting scope. This section identifies which objects are available for scripting, depending on the execution context.

Server-side API

This section lists the APIs that are available and relevant when developing managed beans, listeners or data sources on the server side for a Rich Web Charts component.

Server configuration

In this section you will learn how to configure the Rich Web Charts server.

Styling

Describes the use of cascaded style sheets to control the appearance of the Rich Web Charts.

Rich Web Charts

Describes the Rich Web Charts component.

In this section

Introduction to Rich Web Charts
Identifies the main features of the Rich Web Charts component.

Supported graphical representations
Identifies the Cartesian charts supported by the Rich Web Charts component.

View and data interactions
Describes the available view and data interactions.

JViews Swing Charts and Rich Web Charts - features comparison
Compares the features between JViews Swing Charts and Rich Web Charts components

Requirements
Identifies the requirements necessary to use the Rich Web Charts component.

Introduction to Rich Web Charts

The Rich Web Charts component has the following main characteristics:

- Provide rich and responsive interactions and displays on the Web client going beyond HTML and raster rendering.
- Provide high scalability on the server side.
- Are compatible with IBM® ILOG® JViews Charts. The Rich Web Charts component is able to read project files and style sheets generated by the JViews Charts Designer and to connect to the JViews Charts `IlvDataSource`.

The server-side components are based on the JavaServer™ Faces (JSF) technology and are available through JavaServer Pages (JSP™) tags.

The Rich Web Charts server combines the JSP page content with the style sheet to generate Scalable Vector Graphics (SVG) and JavaScript™ logic that is executed on the Web client to build the chart display and handle user interactions.

The Rich Web Charts client retrieves data and data updates from the server and creates or updates the chart in SVG. Only the chart display is redrawn, the remainder of the page does not need to be refreshed. The user may also continue his work with the chart while data is retrieved in the background.

Interactions with the chart display - such as zooming, scrolling and panning - are executed in SVG and handled directly on the Web client. The absence of server roundtrip and the local rendering in SVG result in an improved user experience, both in terms of look and feel and response time.

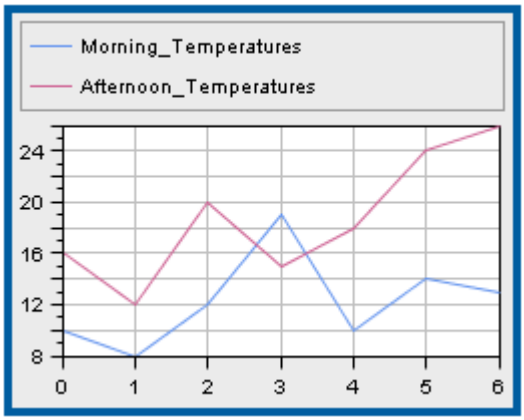
The Rich Web Charts component offers you the following main features:

- *Supported graphical representations*
- *View and data interactions*
- *Data source connectivity*
- *Stylable components*

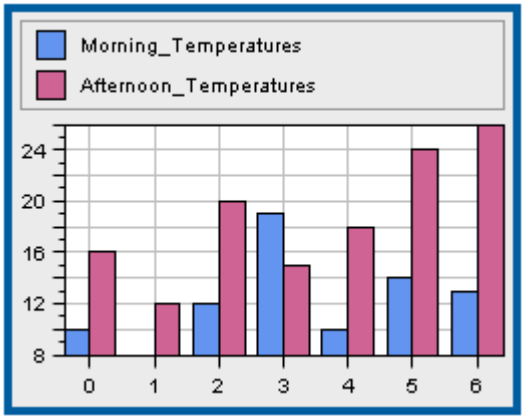
Supported graphical representations

The Rich Web Charts component supports Cartesian charts with seven rendering types and all their subtypes:

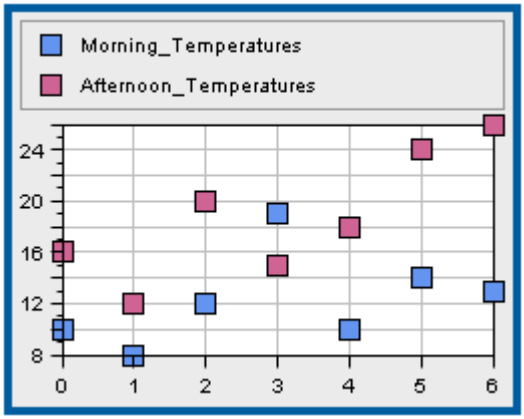
- Polyline



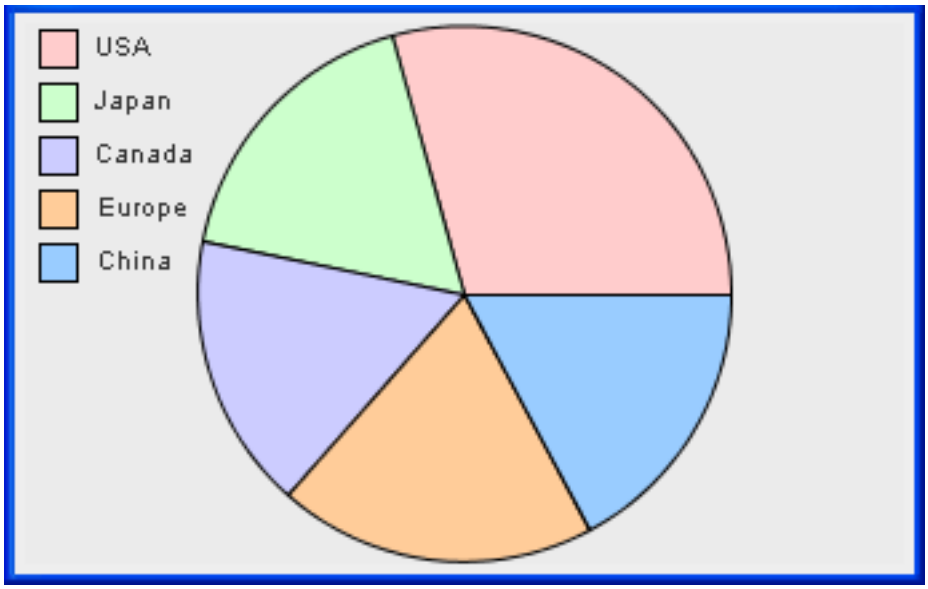
- Bar



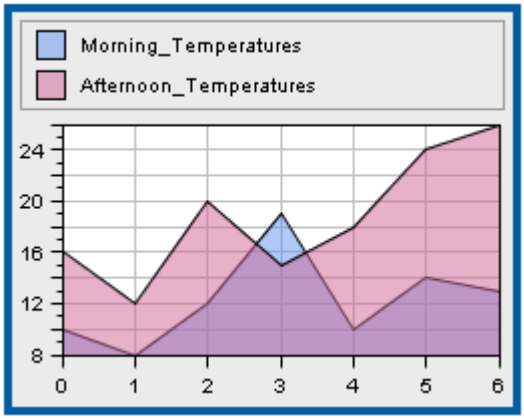
- Scatter


Morning_Temperatures
Afternoon_Temperatures
24
20
16
12
8
0
1
2
3
4
5
6

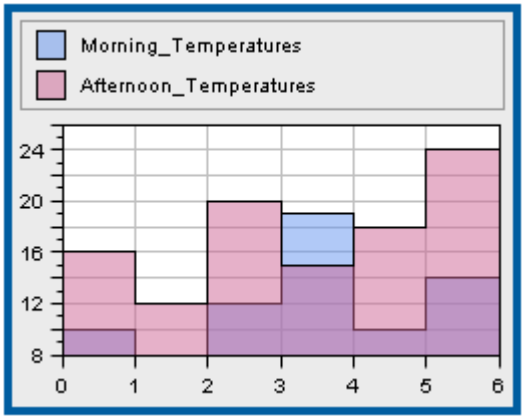
- Pie


USA
Japan
Canada
Europe
China

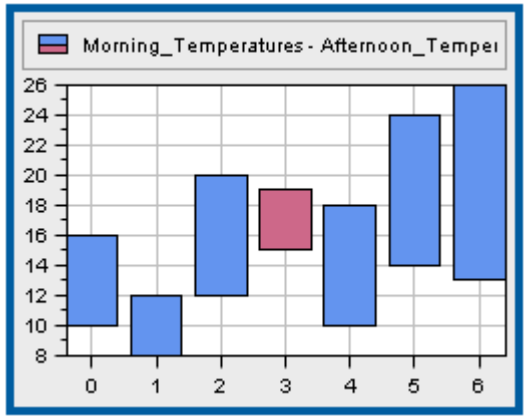
- Area



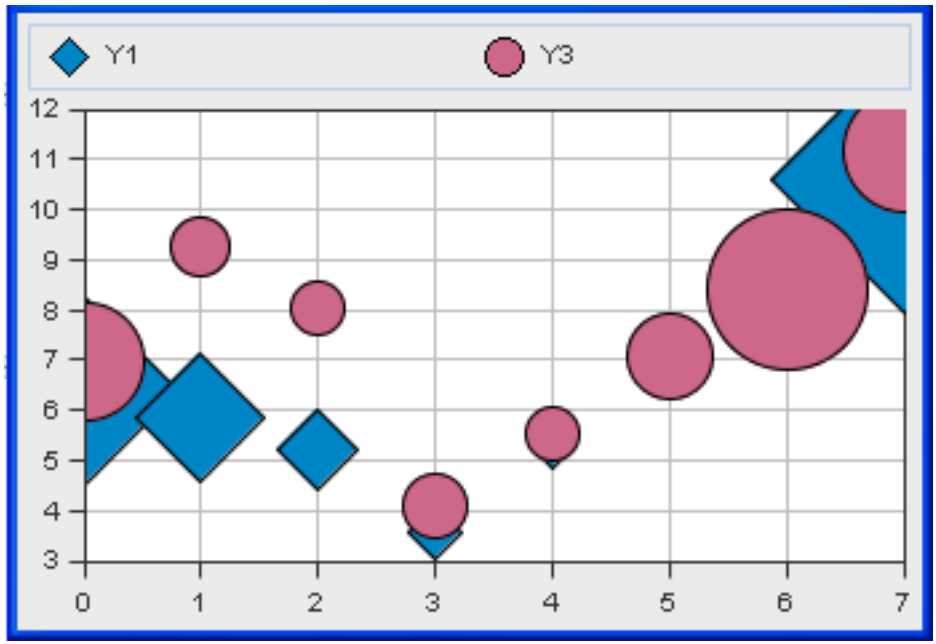
- Stair



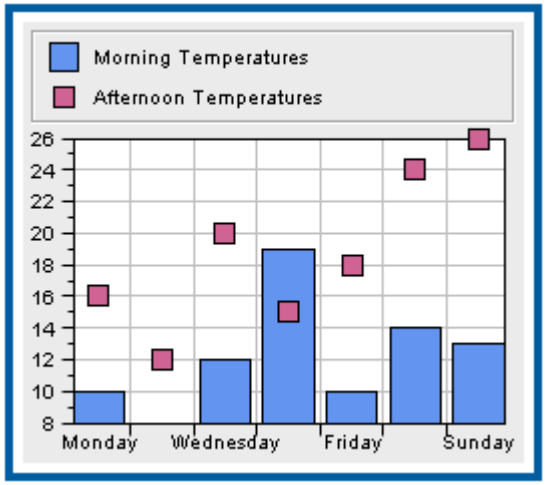
- High-Low



- Bubble



In addition to that, you can combine the different rendering types into a single chart using the so called Combo rendering type.


Morning Temperatures
Afternoon Temperatures
26
24
22
20
18
16
14
12
10
8
Monday
Wednesday
Friday
Sunday

View and data interactions

The following interactions are available on the chart view:

- Zoom in (CTRL+Mouse drag)
- Zoom out (Shift+Mouse drag)
- Scroll (Arrow keys)
- Pan (Secondary mouse button drag)

For details on how to configure the view interactions see *The tag library*.

Data interactions

You can trigger client-side or server-side actions when:

- Hovering a data point (also known as highlighting)
- Clicking a data point (also known as picking)

Client-side actions typically use the client-side JavaScript API to retrieve information on the data involved in the interaction and transform or display additional information about it.

Server-side actions are JSF actions that typically allow you to change the server-side configuration of the chart component, in particular the style sheet and the data source, or to navigate from one page to another one.

For details on how to configure the data interactions see *The tag library*.

Data source connectivity

The Rich Web Charts component can connect to any JViews Charts data source. The server listens to the data source modifications. When requested, the server sends the changes in XML to the client that updates its display accordingly. The client queries data updates either at regular intervals, or on request.

The data source to connect to can be specified:

- by a project file generated with the Designer,
- by an instance of an `IlvDataSource`, referenced by a managed bean property.

For details on how to define the data source to connect to, see *The chart tag*.

Stylable components

The Rich Web Charts component can be configured with JViews Charts style sheets. For details on how to specify the style sheet, see *The chart tag*.

The current version of the Rich Web Charts component has some limitations in its feature set and consequently some of the CSS properties and selectors are not supported.

For more details see *Styling*.

JViews Swing Charts and Rich Web Charts - features comparison

This section compares the features between JViews Swing Charts and Rich Web Charts components. The following table lists the features that are available from JViews Swing Charts when you use the Rich Web Charts component.

The feature list is based on the information provided in the section Basic Concepts in Introducing JViews Charts.

Feature	JViews Swing Charts	Rich Web Charts
Displays	2D, 3D	2D
Data Display	Java2D-based Renderers	SVG-based Renderers
Data Model	`IlvDataSource`	`IlvDataSource`
Type	Cartesian, Radar, Polar, Pie	Cartesian
Representation	Polyline, Bar, Area, Bubble, High Low, Scatter, Stair, Combo, Treemap	Polyline, Bar, Scatter, Combo
Area	Available	Available
Header/Footer	Swing Components in the Chart Component	HTML Tags in the JSP page
Axis	One abscissa, one or several ordinate	One abscissa and one ordinate
Scale	Rectangular, Circular	Rectangular
Grid	Available	Available
Legend	Available	Available
Decoration	Data Indicators, Annotations, Labels, Images	Data Indicators
Drawing Order	Available	Available
Interactors	Zoom, X/Y Scroll, Pan, Highlight-Point, Information-View, Pick Data Points,Local Pan, Local Reshape, Local Zoom, Edit-Point, Action	Zoom, X/Y Scroll, Pan, Highlight-Point, Information-View, Pick Data Points

For details on the exact supported features set, refer to *Styling*.

Requirements

This section contains the requirements necessary to use the Rich Web Charts component.

Client

The Rich Web Charts component requires on the client an HTTP 1.1 user agent that is able to interpret HTML 4.0 and SVG 1.0 or higher. The user agent must support `embed` tag to include SVG and provides Ajax functionalities through an `XMLHttpRequest` object.

The Rich Web Charts component is supported on the following user agents:

- Internet Explorer 6.0 or 7.0 with Adobe™ SVG Viewer 3.x
- Firefox 2.0 or 3.0

Server

The Rich Web Charts component is based on Servlets and JavaServer Faces technologies and requires an HTTP 1.1 Web Application Server that supports Servlet version 2.3 and JSF version 1.1 or later.

The Rich Web Charts component is supported on all servers that fulfill the above requirements, it has been tested and is working with the following JSF implementations:

- Sun JSF 1.1 Reference Implementation
- Sun JSF 1.2 Reference Implementation
- MyFaces 1.1.5

Proxy

The Rich Web Charts component requires the use of HTTP 1.1, and will thus support only HTTP 1.1 proxies.

Architecture overview

Provides an overview of the architecture of a Rich Web Charts application.

In this section

Introduction
Explains the organization of a Rich Web Charts application.

Run-time process flow
Identifies the actions performed by the Web client at run time.

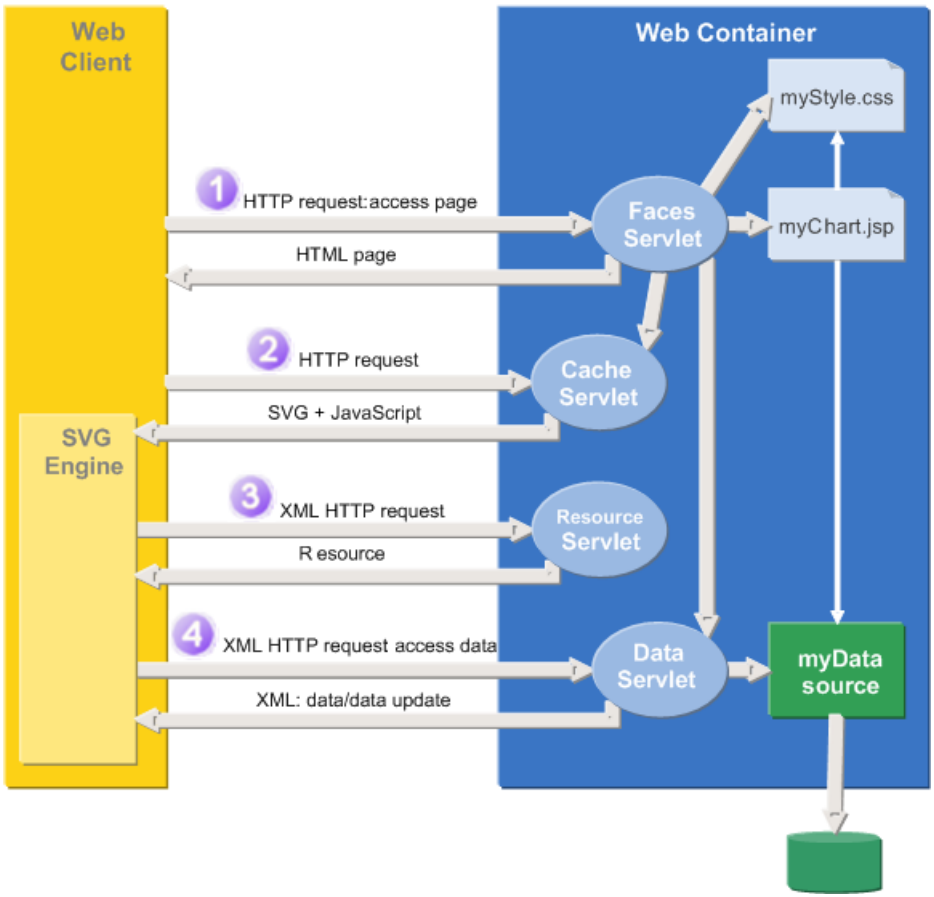
Main classes
Identifies the main classes involved on the Web server and Web client.

Introduction

A Rich Web Charts application is organized according to a two-tier or three-tier approach and is composed as follows:

- the Web client, which stores the chart data, performs all the graphical rendering and handles user interactions,
- the Web container, which provides the Web client with a dynamic SVG document configured from JSP tags and a style sheet. It also provides the client with the chart data from an `IlvDataSource`,
- optionally a back-end, which provides the contents of the data source to the `IlvDataSource`.

Rich Web Charts application shows the different elements of a Rich Web Charts application:



Rich Web Charts application

The Web container holds the definition of the Rich Web Charts application:

- `myStyle.css` is an IBM® ILOG® JViews Charts style sheet.
- `myDataSource` is an `IlvDataSource`, that may optionally retrieve its contents from an external data source.
- `myChart.jsp` is a JSP page containing a `chart` tag that references `myStyle.css` and `myDataSource`.

The Web container also contains the following servlets:

- Faces Servlet – implementing the JSF specifications, it renders the JSP pages as HTML, SVG and JavaScript and handles the server-side actions.
- Cache Servlet – is an internal cache required to workaround Web client specific issues. It provides the Web client with a dynamic SVG document.
- Resource Servlet – provides the Web client with server-side resources such as JavaScript code.
- Data Servlet – maintains an history of data source changes. It handles client requests for data and data updates.

Run-time process flow

At run time, the Web client performs the following actions:

- Accesses the JSP page that contains the chart  . In response, the server:
  - builds the HTML page that is returned to the client,
  - renders the chart tags and the style sheet as an SVG document enriched with JavaScript code. This document is then passed to the Cache Servlet for future retrievals made by the client.
- Retrieves the SVG document from the Cache Servlet  and the JavaScript resources from the Resource Servlet  .
- Renders the chart component and starts handling user interactions.
- Retrieves the data source contents from the Data Servlet  and renders the data in SVG. This is performed asynchronously.

From now on, the Web client:

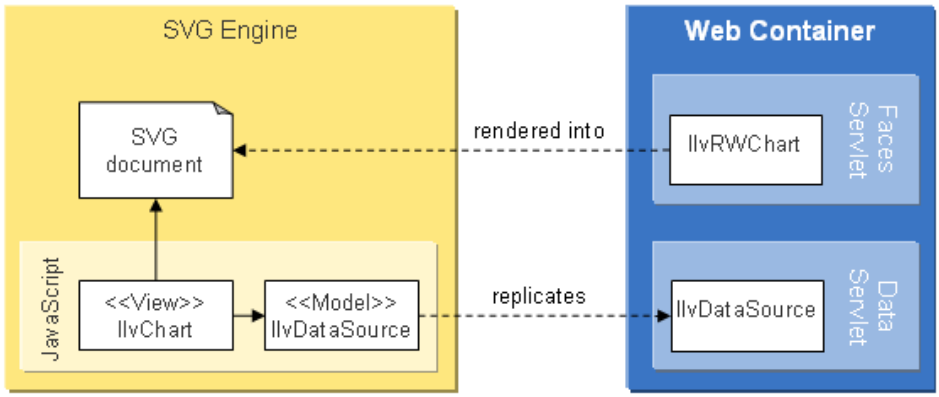
- Periodically retrieves the data source changes from the Data Servlet  and renders the changes in SVG. This is performed asynchronously.
- Handles user interaction with the chart view and updates the view - with no round-trip to the server.
- Handles user interactions with the chart data.

Note: Regarding interactions with chart data:

- Client-side actions are performed locally.
- Server-side actions are performed according to the JSF specifications, submitting the HTML form. This can either be done synchronously following the whole JSF lifecycle or asynchronously using an Ajax submit if the submitted action does not require any JSF component to re-render.

Main classes

Class diagram shows the main classes involved on the Web server and Web client.



Class diagram

On the Web server:

- `IlvRWChart` is the JSF UI Component that describes the chart. It is rendered into an SVG document.
- `IlvDataSource` contains the data model used by the chart. It is replicated on the Web client side.

On the Web client:

- `IlvDataSource` contains the data model used by the chart. It is a replication of the server-side `IlvDataSource`.
- `IlvChart` is the view part of the chart component. It renders the data model in the SVG document and handles the user interactions.

For more information on the client-side classes, see *Client-side API*. For more information on the server-side classes, see *Server-side API*.

Getting started

This section provides a walkthrough of the basic steps required to create and deploy a Rich Web Charts application.

In this section

Creating a JViews Charts project with the Designer
Presents an overview of how to use the Designer to create a JViews Chart project.

Creating a JSP page using Rich Web Charts
Presents an overview of how to create a JSP page using Rich Web Charts.

Setting up a data source
Presents an overview of the use of `IlvDataSource` for data.

Defining client and server-side actions in response to user interactions
Presents an overview of how to define client and server-side actions to execute in response to events.

Configuring the client update interval
Presents an overview of the configuration of the client update interval.

Deploying the Web application
Presents an overview of the configuration of the Web application deployment.

Creating a JViews Charts project with the Designer

By using the Designer, you can define the styling and the data connection of your chart in a JViews Charts project (see Using the Designer). When you save the project you get a project file (with the `.icpr` extension) and a style sheet file (with the `.css` extension). These files are used in the next step to configure the Rich Web Charts component.

The Rich Web Charts component supports a subset of the features available in the Designer. For more details, see *Styling*.

Creating a JSP page using Rich Web Charts

Use a Web application development environment to create a JSP page. The page may mix one or more Rich Web Charts components with other third party JSF components.

The `chart` tag defines an area where a chart component is rendered along with the styling and data source to use. It may reference the project file previously created, as follows:

```
?request=image&format=PNG&width=400&height=300&action=LegendVisibili
tyAction(true)
<jvrc:chart width="450px" height="300px" value="project.icpr"/>
```

For more details on how to add a chart component in a JSP page, see *Add a Rich Web Charts to a JSP page*.

You can use optional interactors tags to customize the view interactions, such as zooming, scrolling and panning. For details on how to use these tags, see *Set up interactions*.

Setting up a data source

Optionally, instead of getting its data connection information from a Designer project file, the Rich Web Charts component can be configured to access an instance of `IlvDataSource`. You can use this option in one of the following cases:

- you need more flexibility to define where data is queried,
- you need other components of the JSP page to access the data source, to modify its contents,
- you use a custom `IlvDataSource`.

In these cases, the instance of `IlvDataSource` must be provided through a server-side managed bean property and bound to the `jvrc:chart` tag, as follows:

```
<jvrc:chart width="450px" height="300px"
            value="#{myServerSideBean.dataSource}"
```

For more details on how to reference an `IlvDataSource`, see *Add a Rich Web Charts to a JSP page*.

For more details on the data source API, see *Data source*.

Defining client and server-side actions in response to user interactions

The `highlightInteractor` and `pickInteractor` tags fire events when the user respectively goes over a data point or selects it. In response to these events, you can define client-side actions that will be executed.

Additionally, with the `jvrc:pickInteractor` tag you can register JSF server-side actions to be executed in the JSF lifecycle.

For more details, see *Trigger a client-side action when picking a data point* and *Trigger a server-side action when picking a data point*.

Configuring the client update interval

The data source contents may change at run time. In this case, you must:

1. define the client update interval and set the corresponding attribute in the `chart` tag,
2. set the server configuration accordingly.

For more details, see *Configure the update interval* and *Configuring the data source replication*.

Deploying the Web application

You need to configure the deployment of the Web application to:

- register the Rich Web Charts Servlets,
- declare the managed beans, if any,
- set up other server configuration parameters.

For more details on how to set up the Web application deployment descriptor for the Rich Web Charts, see *Server configuration*.

Now the application is ready to be deployed and accessed from an HTML/SVG user agent.

How to...

This section shows you how to use more of the features of the Rich Web Charts application.

In this section

Add a Rich Web Charts to a JSP page
Describes how to add a Rich Web Charts component to a JSP page.

Configure the update interval
Describes how to configure the update interval.

Set up interactions
Describes how to add a new interaction.

Change the style sheet at run time
Describes how to change the style sheet used by the Rich Web Charts component at run time.

Update Chart data without refreshing the whole HTML page
Describes how to update data while avoiding a refresh of the full HTML page.

Change the color of the data point under the mouse
Describes how to change the color of a data point under the mouse pointer.

Trigger a client-side action when picking a data point
Describes how to trigger a client-side action when picking a data point by installing a pick interactor on your Rich Webs Charts component.

Trigger a server-side action when picking a data point
Describes how to trigger a server-side action when picking a data point by installing a pick interactor on your Rich Webs Charts component.

Add a Rich Web Charts to a JSP page

The Rich Web Charts component is available as a JSF component and thus can be used inside a JSP page.

For example, if you have the following JSP page using JSF:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<html>
  <body>
    <f:view>
      <h:form id="form">
       <h:commandButton value="Submit"
 action="#{myServerSideBean.performAction}">
      </h:form>
    </f:view>
  </body>
</html>
```

you can add a Rich Web Charts component by doing the following:

1. include the following Tag Library import statement at the top of the JSP file:

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

2. add a `chart` tag inside the `h:form` tag:

   ```
   <jvrc:chart id="thechart" width="450px" height="300px"
               value="project.icpr"/>
   ```

The HTML page generated from this JSP page will contain the chart described by the project.icpr file displayed as SVG.. The chart will be displayed in an area that is 450 pixel wide by 300 pixel tall.

Alternatively, the `value` attribute can reference an instance of `IlvDataSource` available on the server-side Bean as follows:

```
<jvrc:chart id="thechart" width="450px" height="300px"
            value="#{myServerSideBean.dataSource}"
            stylesheet="style.css"/>
```

As you can see, the JViews Charts style sheet used to style the chart is specified in a `stylesheet` attribute.

For the detailed description of the attributes on the `jvrc:chart` tag, see *The tag library*.

Configure the update interval

In case the `IlvDataSource` (either referenced directly by the `value` attribute or through an `.icpr` project) values on the Web server are regularly updated, you may want that your Rich Web Charts application automatically displays the new values.

For that, you have to add an `updateInterval` attribute on the `chart` tag specifying the time in seconds the client has to wait between two requests sent to the server to update data. For more details on the `updateInterval` attribute, see *The tag library*.

In order to obtain the best performances, check the server configuration to fit the update interval on the Web client. For more details, see *Configuring the data source replication*.

Set up interactions

As illustrated in the section *The tag library*, several interactors are available on the `chart` tag.

If you want to provide a new interaction, you need to add the corresponding tag as a child of your `jvrc:chart` tag, as follows:

```
<jvrc:chart id="thechart" width="450px" height="300px"
            value="project.icpr">
   <jvrc:zoomInteractor/>
</jvrc:chart>
```

The `zoomInteractor` tag allows the user to zoom in or out the chart area.

The same procedure can be followed for other interactors. If there is a conflict between two interactors, the last interactor defined will take precedence.

In any case, interactors defined in the JSP page take precedence over interactors defined in the style sheet associated with the `chart` tag.

Change the style sheet at run time

To change the style sheet used by the Rich Web Charts component at run time, you can simply rely on the JSF mechanism by using a `chart` tag configured as follows:

```
<jvrc:chart id="thechart" width="450px" height="300px"
            value="#{myServerSideBean.dataSource}"
            stylesheet="#{myServerSideBean.stylesheet}"/>
```

The server-side Bean would look like the following:

```
public class MyServerSideBean {
  private String stylesheet;

  public String getStylesheet() {
    return stylesheet;
  }

  public void setStylesheet(String sheet) {
    stylesheet = sheet;
  }
}
```

In the JSF framework, such a server-side Bean can be registered as a managed bean in the `faces-config.xml` file of your application as follows:

```
<managed-bean>
  <description>myServerSideBean</description>
  <managed-bean-name>myServerSideBean</managed-bean-name>
  <managed-bean-class>mypackage.MyServerSideBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

On the JSP page, you can use a JSF combo-box component to change the style sheet so that the new style is reflected on the chart component:

```
<h:selectOneMenu value="#{myServerSideBean.stylesheet}">
  <f:selectItem itemLabel="css 1"
                itemValue="/data/stylesheet1.css" />
  <f:selectItem itemLabel="css 2"
                itemValue="/data/stylesheet2.css" />
</h:selectOneMenu>
```

When the page is submitted, the JSF combo-box updates the style sheet property of the Bean. During the rendering process, the chart component will use the updated value of this property.

Update Chart data without refreshing the whole HTML page

In *Change the style sheet at run time* we have seen how to change a parameter of the Chart using JSF. However, in some cases you may want to avoid going through the whole JSF lifecycle and prevent the refresh of the whole HTML page.

Since chart data is provided by a separate Servlet (see *Architecture overview*), it is possible to avoid the full refresh when a JSF action modifies the contents of the `IlvDataSource` instance on the server. For example, this is the case when you use the `updateInterval` attribute on `chart` tag. However, this automatic refresh does not respond to a client request and no new information can be sent from the client to the server during that process.

When you want to send information from the client to the server, asynchronously and without refreshing the whole HTML page, you have to use the JSF framework by submitting the information using an Ajax methodology instead of the regular JSF submit.

For example, if you want to tell the server to add a new `IlvDataSet` to the displayed `IlvDataSource` and then update the client representation, you can proceed as follows:

Have in the JSP page a JSF component that allows the user to add a new data set:

```
<h:commandButton action="#{myServerSideBean.addDataSet}"
                 value="Add a Data Set "/>
```

with `MyServerSideBean` defined as follows:

```
public class MyServerSideBean {
  private IlvDataSource dataSource = new IlvDefaultDataSource();

  public IlvDataSource getDataSource() {
    return dataSource;
  }

  void public addDataSet() {
    dataSource.addDataSet(new IlvDefaultDataSet());
  }
}
```

However, if you leave the button as-is, the form will be submitted going through the whole lifecycle. During the lifecycle, the server-side Bean adds the `IlvDataSet` and the JSF will regenerate the whole HTML page. In order to do it through an Ajax request, you must follow these four steps:

1. Add a Faces listener to the `faces-config.xml` of your application. This listener handles Ajax requests:

   ```
   <lifecycle>
      <phase-listener>
   ilog.views.chart.rwc.faces.internal.lifecycle.IlvRWAjaxRequestHandler
      </phase-listener>
   </lifecycle>
   ```

2. Add a client-side listener to the button (that adds the data set) to submit the form using the Ajax methodoly when the user clicks the button:

```
<h:commandButton …
   onclick="IlvAjaxUtil.submitForm(this, updateDataSource) "/>
```

3. Make sure the client-side data is updated after the form has been submitted (and thus the `IlvDataSet` has been added on the server):

```
<script>
  // refresh the data of the <jvrc:chart> with the id
  // 'thechart'
  function updateDataSource() {
     thechart.updateDataSource();
  }
</script>
```

4. Make sure to disable the standard behavior of the button for submitting the form by binding it to a special instance of `HtmlCommandButton` class that does not automatically submit the form:

```
<h:commandButton …
   binding="#{myServerSideBean.noSubmitButton}"/>
```

with `MyServerSideBean getNoSubmitButton` method returning an instance of `IlvFacesNoSubmitButton`:

```
public class MyServerSideBean {
  // …
  private HtmlCommandButton button =
     new IlvFacesNoSubmitButton();

  public HtmlCommandButton getNoSubmitButton() {
     return button;
  }
  public void setNoSubmitButton(HtmlCommandButton b) {
     button = b;
  }
}
```

This process allows the `IlvDataSource` instance on the server to be updated with a new `IlvDataSet` on a client-side request. Then, the client side is updated with the new data, without refreshing the whole set of components.

Change the color of the data point under the mouse

If you need to change the color of the data point under the mouse pointer you can use a `highlightInteractor` tag, as follows:

```
<jvrc:highlightInteractor style="fill:red"/>
```

This allows you to set the primary color of the data point under the mouse pointer to red when going over it. To set the secondary color you can use the stroke property instead or in conjunction with the fill property.

The `style` attribute accepts the following SVG properties: fill, stroke, fill-opacity, stroke-opacity, stroke-width and all properties that allow you to configure the stroke.

For information on other `highlightInteractor` tag attributes, see *The tag library*.

Trigger a client-side action when picking a data point

To trigger a client-side action when picking a data point, you need to install a pick interactor on your Rich Web Charts component.

```
<jvrc:chart id="thechart" width="450px" height="300px"
            value="project.icpr">
  <jvrc:pickInteractor
     onpick="alert(pickedPoint.getXData()+' '
                   +pickedPoint.getYData())"/>
</jvrc:chart>
```

In the onpick event handler, the `pickedPoint` object is available to get information on the picked point.

See *Client-side API* for more details on the API available for client-side event handlers.

Trigger a server-side action when picking a data point

To trigger a server-side action when picking a data point, you need to install a pick interactor on your Rich Web Charts component.

```
<jvrc:chart id="thechart" width="450px" height="300px"
            value="project.icpr">
  <jvrc:pickInteractor
       actionListener="#{myServerSideBean.actionPerformed}"/>
</jvrc:chart>
```

The `actionPerformed` is a method of `myServerSideBean` that takes as parameter an `RWChartInteractionEvent` instance.

The following code sample shows an example of the `actionPerformed` method:

```
public void actionPerformed(RWChartInteractionEvent event) {
  IlvDataSetPoint point = event.getPoint();
  IlvDataSet dataSet = point.getDataSet();
  IlvDataSource dataSource = event.getChart().getDataSource();
  // Some action
}
```

Alternatively, the `listening` method can take `javax.faces.event.ActionEvent` type as parameter in order to be shared with other JSF components. For example:

```
public void actionPerformed(ActionEvent event) {
  // does some action common to all components
  if (event instanceof RWChartInteractionEvent) {
    RWChartInteractionEvent evt = (RWChartInteractionEvent)event;
    // does some additional action for Chart
  }
}
```

The server-side actions set above will be triggered during a full JSF lifecycle request, leading to the full refresh of the HTML page. If the server-side action does not need to refresh the client directly, you can perform the server-side action using the Ajax methodology instead of the regular JSF submit. To do this, proceed as follows:

1. Add a client-side action that will be in charge of submitting the action to the server in an asynchronous manner:

   ```
   <jvr:pickInteractor … onpick="IlvAjaxUtil.submitForm(this, oncompleted)
   "/>
   ```

   The `oncompleted` parmeter is optional. This can be a function in charge of doing some client-side actions when the server-side action is executed (see *Update Chart data without refreshing the whole HTML page*).

2. Add a Faces listener to the `faces-config.xml` file of your application. This listener handles Ajax requests:

```
<lifecycle>
  <phase-listener>
ilog.views.chart.rwc.faces.internal.lifecycle.IlvRWAjaxRequestHandler
  </phase-listener>
</lifecycle>
```

   Server-side Ajax actions can also be fired when moving the mouse cursor over the chart by using the same methodology on the `highlightInteractor`.

For information on other `pickInteractor` tag attributes see *The tag library*.

The tag library

Describes the JSP tag library of the Rich Webs Charts component.

In this section

Introduction
Introduces the JSP tag library.

The chart tag
Describes the attributes and default values of the chart tag.

The highlightInteractor tag
Describes the attributes and default values of the highlightInteractor tag.

The infoViewInteractor tag
Describes the attributes and default values of the infoViewInteractor tag.

The panInteractor tag
Describes the attributes and default values of the panInteractor tag.

The pickInteractor tag
Describes the attributes and default values of the pickInteractor tag.

The xScrollInteractor tag
Describes the xScrollInteractor tag.

The yScrollInteractor tag
Describes the yScrollInteractor tag.

The zoomInteractor tag
Describes the attributes and default values of the zoomInteractor tag.

Introduction

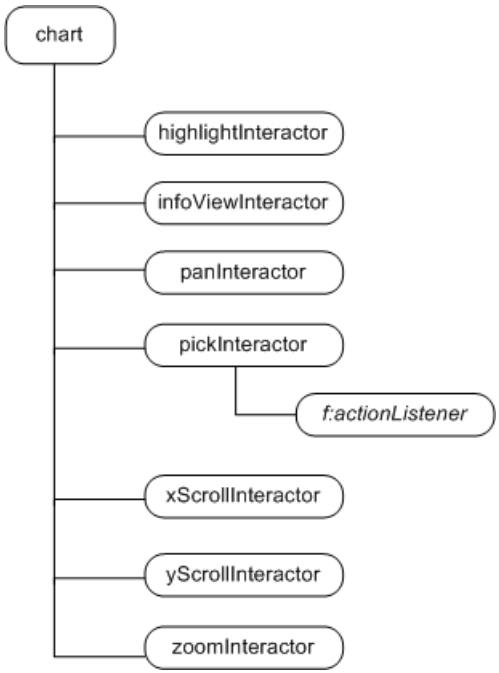
The Rich Web Charts component consists in a JSP tag library. If you want to use this tag library in your page, you need to insert the following code:

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

Note that `jvrc` is the declared prefix. If you want to use a tag of this library in the JSP page, you must prefix it by `jvrc`:

Using a tag of the declared tag library with the prefix jvrc

```
<jvrc:chart [...] />
```



Rich Web Charts tag hierarchy

Note: f:actionListener is a tag of the JavaServer Faces reference implementation.

In the following pages you are going to find more detailed information about the Rich Web Charts tags:

- *The chart tag*

- *The highlightInteractor tag*
- *The infoViewInteractor tag*
- *The panInteractor tag*
- *The pickInteractor tag*
- *The xScrollInteractor tag*
- *The yScrollInteractor tag*
- *The zoomInteractor tag*

The chart tag

This tag declares a chart which is using SVG to render itself inside an HTML page. Interactors can be registered as subtags to allow interactions on this chart.

Name	Required	Default value
height	Yes	Not applicable
width	Yes	Not applicable
keepVisibleWindow	No	`true`
stylesheet	No	`null`
updateInterval	No	0
value	Yes	Not applicable
resizingPolicy	No	DEFAULT_RESIZING_POLICY

height

Specifies the height of the chart. The value can be postfixed by a CSS unit.

Typical height attribute uses

```
<jvrc:chart height="300" [...] />
<jvrc:chart height="300px" [...] />
<jvrc:chart height="100%" [...] />
```

This attribute is required.

width

Specifies the width of the chart. The value can be postfixed by CSS unit.

Typical width attribute uses

```
<jvrc:chart width="300" [...] />
<jvrc:chart width="300px" [...] />
<jvrc:chart width="100%" [...] />
```

This attribute is required.

keepVisibleWindow

Specifies whether the visible window should be reused by the client when a new JSF request occurs. The default value is `true`. When a JavaServer Faces request is issued, for example when a server-side action is triggered, the current visible window is sent to the server. When this attribute is set to `true`, this visible window is sent back to the client to retrieve the same

view state before the request. In some cases, for example when the data source has changed, the visible window must not be sent back to the client. In this case, set the attribute to `false`.

A more flexible method consists in binding this property to a Bean property to dynamically change its value.

Bind the keepVisibleWindow property

```
<jvrc:chart [...] keepVisibleWindow="#{bean.keepVisibleWindow}" />
```

stylesheet

Specifies the URL to a chart style sheet. If the `value` attribute contains a URL to a Designer project that already defines a style sheet, the style sheet specified by this attribute will override the style sheet specified in the project.

If the `value` attribute contains a binding to an `IlvDataSource` instance, the `stylesheet` attribute must be set otherwise the chart uses the default styling.

updateInterval

Specifies the interval in seconds between two data update requests sent to the server. When the value is 0, it means that only the initial data request will be made. The default value is 0.

value

Specifies the URL to a Designer project or a value expression to an `IlvDataSource` instance.

Using a Charts Designer project

```
<jvrc:chart [...] value="data/chart.icpr" />
```

As you can see in *Using a Charts Designer project*, the chart uses the data source and the style sheet specified in the project.

Using an IlvDataSource instance

```
<jvrc:chart [...] value="#{bean.datasource}" />
```

The chart uses the specified data source. Use the stylesheet attribute to specify a style sheet for the chart. `bean` is a JavaServer Faces managed bean and `datasource` a property of this bean of type `IlvDataSource`.

This attribute is required.

resizingPolicy

Allows you to specify which resizing policy will be used on the chart when the browser window is resized and the chart size depends on the browser window (specified with percentages, see the description of the *width* attribute). The default policy (DEFAULT_RESIZING_POLICY) is such that the visible range remains the same regardless the size of the chart. The RANGE_RESIZING_POLICY value makes sure that when the chart

is resized the visible range is reduced accordingly. This is valid if the chart is not in `autoVisibleRange` mode (see *Axis properties*).

This attribute is not required.

The highlightInteractor tag

This tag declares an interactor that fires client or serve-side actions when the mouse cursor is over a point. For client-side actions, the JavaScript code contained in the `onhighlight` attribute will be called when the mouse is over a data point according to the mode set in the `highlighMode` attribute. For server-side actions configured using the `action` or `actionListener` attributes, only asynchronous submissions are supported to avoid blocking user interactions. Refer to *Trigger a server-side action when picking a data point* to see how to perform an asynchronous server-side action.

Name	Required	Default value
action	No	`null`
actionListener	No	`null`
immediate	No	`false`
highlightMode	No	`HIGHLIGHT_POINT`
marker	No	`null`
onhighlight	No	`null`
pickingMode	No	`NEARESTPOINT_PICKING`
style	No	`null`

action

Method binding representing the application action to invoke when this component is activated by the user. The expression must evaluate to a public method that takes no parameters and returns a `String` (the logical outcome) which is passed to the navigation handler for this application.

actionListener

Method binding representing an action listener method that will be notified when this component is activated by the user.

The expression must evaluate to a public method that takes an `javax.faces.event.ActionEvent` parameter, with a return type of `void`.

immediate

This flag indicates that if this component is activated by the user, notifications should be delivered immediately to the interested listeners and actions, that is, during the Apply Request Values phase, rather than waiting the Invoke Application phase.

The default value is `false`.

highlightMode

Set the highlight mode of this interactor.

If set to `HIGHLIGHT_POINT`, an event is sent whenever the mouse enters and leaves a display point.

If set to `HIGHLIGHT_SERIES`, an event is sent only if the new display point does not belong to the same data set as the previously highlighted point.

The default mode is `HIGHLIGHT_POINT`.

marker

The marker to be displayed on the chart data point or series when the mouse pointer is over a data point. Valid values are `CIRCLE`, `CROSS`, `DIAMOND`, `PLUS`, `TRIANGLE`, `SQUARE`.

onhighlight

The JavaScript code executed on the client when a point is highlighted.

pickingMode

Set the picking mode of this interactor.

The picking mode defines how to retrieve the picked data point. Two modes are supported:

- `NEARESTPOINT_PICKING`: retrieves the data point closest to the picking point.
- `ITEM_PICKING`: retrieves the data point that contains the picking point.

The default mode is `NEARESTPOINT_PICKING`.

style

The inline SVG style applied on the client side to the chart data point or series when an event is fired by the interactor.

Use of style attribute to highlight the point with red color

```
<jvrc:chart [...] >
  <jvrc:highlightInteractor style="color:red" />
</jvrc:chart>
```

The `style` attribute accepts the following SVG properties: fill, stroke, fill-opacity, stroke-opacity, stroke-width and all properties that allow you to configure the stroke.

For more details on these SVG properties, you can refer to the following URLs:

- *http://www.w3.org/TR/SVG10/painting.html#FillProperties*
- *http://www.w3.org/TR/SVG10/painting.html#StrokeProperties*

The infoViewInteractor tag

This tag declares an interactor that displays an information tooltip when the mouse pointer is over a chart data point.

Name	Required	Default value
followMouse	No	`false`
pickingMode	No	`NEARESTPOINT_PICKING`

followMouse

Specifies whether the info view follows the mouse moves. The default value is `false`.

pickingMode

Set the picking mode of this interactor.

The picking mode defines how to retrieve the picked data point. Two modes are supported:

- NEARESTPOINT_PICKING: retrieves the data point closest to the picking point.
- ITEM_PICKING: retrieves the data point that contains the picking point.

The default mode is `NEARESTPOINT_PICKING`.

The panInteractor tag

This tag declares an interactor to pan the chart area using the right mouse button.

Name	Required	Default value
xAxisAllowed	No	`true`
yAxisAllowed	No	`false`

xAxisAllowed

Specifies whether panning along the X-axis is allowed. The default value is `true`.

yAxisAllowed

Specifies whether panning along the Y-axis is allowed. The default value is `false`.

The pickInteractor tag

This tag declares an interactor that fires an event when a point is picked using the left mouse button.

This tag declares an interactor that fires client or server-side actions when a point is picked using the left mouse button. For client-side actions, the JavaScript code contained in the `onpick` attribute will be called when a point is picked. The server-side actions configured using the `action` or `actionListener` attributes are submitted by default in a full JSF lifecycle. If asynchronous submission is needed, see *Trigger a server-side action when picking a data point*.

Name	Required	Default value
action	No	`null`
actionListener	No	`null`
immediate	No	`false`
marker	No	`null`
onpick	No	`null`
pickingMode	No	`ITEM_PICKING`
style	No	`null`

action

Method binding representing the application action to invoke when this component is activated by the user. The expression must evaluate to a public method that takes no parameters and returns a `String` (the logical outcome) which is passed to the navigation handler for this application.

actionListener

Method binding representing an action listener method that will be notified when this component is activated by the user.

The expression must evaluate to a public method that takes an `javax.faces.event.ActionEvent` parameter, with a return type of `void`.

immediate

This flag indicates that if this component is activated by the user, notifications should be delivered immediately to the interested listeners and actions, that is, during the Apply Request Values phase, rather than waiting the Invoke Application phase.

The default value is `false`.

marker

Specifies the marker to set on the client side on the chart data point or series when an event is fired by the interactor.

onpick

The JavaScript code executed on the client when a point is picked.

pickingMode

Set the picking mode of this interactor.

The picking mode defines how to retrieve the picked data point. Two modes are supported:

- NEARESTPOINT_PICKING: retrieves the data point closest to the picking point.
- ITEM_PICKING: retrieves the data point that contains the picking point.

The default mode is ITEM_PICKING.

style

Specifies the inline SVG style applied on the client side to the chart data point or series when an event is fired by the interactor.

Use of style attribute to highlight the picked point with red color

```
<jvrc:chart [...] >
  <jvrc:pickInteractor style="color:red" />
</jvrc:chart>
```

The `style` attribute accepts the following SVG properties: fill, stroke, fill-opacity, stroke-opacity, stroke-width and all properties that allow you to configure the stroke.

For more details on these SVG properties, you can refer to the following URLs:

- *http://www.w3.org/TR/SVG10/painting.html#FillProperties*
- *http://www.w3.org/TR/SVG10/painting.html#StrokeProperties*

The xScrollInteractor tag

This tag declares an interactor that handles the scroll along the x-axis. The left-arrow key scrolls the chart to the right. The right-arrow key scrolls the chart to the left.

The yScrollInteractor tag

This tag declares an interactor that handles the scroll along the y-axis. The up-arrow key scrolls the chart down. The down-arrow key scrolls the chart up.

The zoomInteractor tag

This tag declares an interactor to zoom in (CTRL+mouse drag) or zoom out (Shift+mouse drag) a chart area.

Name	Required	Default value
xAxisAllowed	No	`true`
yAxisAllowed	No	`false`
zoomOutAllowed	No	`true`

xAxisAllowed

Specifies whether zooming along the x-axis is allowed. The default value is `true`.

yAxisAllowed

Specifies whether zooming along the y-axis is allowed. The default value is `false`.

zoomOutAllowed

Specifies whether zooming out is allowed. The default value is `true`.

Client-side API

In response to user actions on the client-side, either on the chart component or any other JSF component, it is possible to use some JavaScript code to perform operations. For that purpose the Rich Web Charts client provides some JavaScript objects that are available in the scripting scope. This section identifies which objects are available for scripting, depending on the execution context.

In this section

Globally available objects
Identifies the globally available JavaScript objects.

Objects available in onhighlight event handler
Identifies the JavaScript objects available in the `onhighlight` event handler.

Objects available in onpick event handler
Identifies the JavaScript objects available in the `onpick` event handler.

Globally available objects

For each `chart` tag in the JSP page, a corresponding JavaScript variable of type `IlvChart` is created. This variable takes the name of the value of the `id` attribute.

The variable is defined and available before the onload event is fired.

For example, if you have:

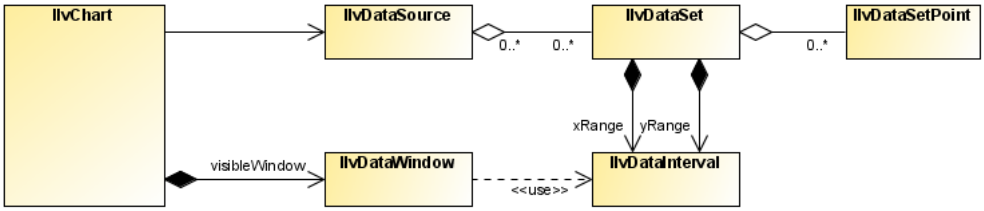
```
<jvrc:chart id="thechart" … />
```

you can access the `IlvChart` instance as follows:

```
thechart.scroll(10, 0);
```

In this particular example, you ask the `IlvChart` instance to scroll the abscissa range by 10, which means that if that axis was previously ranging from 0 to 20, it is now ranging from 10 to 30.

The `IlvChart` instance allows you to scroll, zoom in/out the area, ask for an update of the chart data.



Globally available objects on the client side

Objects available in onhighlight event handler

The following variables are available in the context of the onhighlight event handler of the `highlightInteractor` tag:

Variable name	Type	Description
`highlightedPoint`	`IlvDataSetPoint`	The data point whose highlight state was modified.
`isHighlighted`	Boolean	The new data point highlight state.
`evt`	`UIEvent`	The SVG event at the origin of the action.

The following example shows how to display a dialog box with the y value of the data point under the mouse pointer:

```
<jvrc:highlightInteractor
   onhighlight="if (isHighlighted)
                            alert(highlihlightedPoint.getYDate())"/>
```

Objects available in onpick event handler

The following variables are available in the context of the onpick event handler of the `pickInteractor` tag:

Variable name	Type	Description
`pickedPoint`	`IlvDataSetPoint`	The data point that was picked.
`evt`	`UIEvent`	The SVG event at the origin of the action.

Server-side API

This section lists the APIs that are available and relevant when developing managed beans, listeners or data sources on the server side for a Rich Web Charts component.

In this section

JSF UI components
Lists the JSF tags and the corresponding JSF UI components.

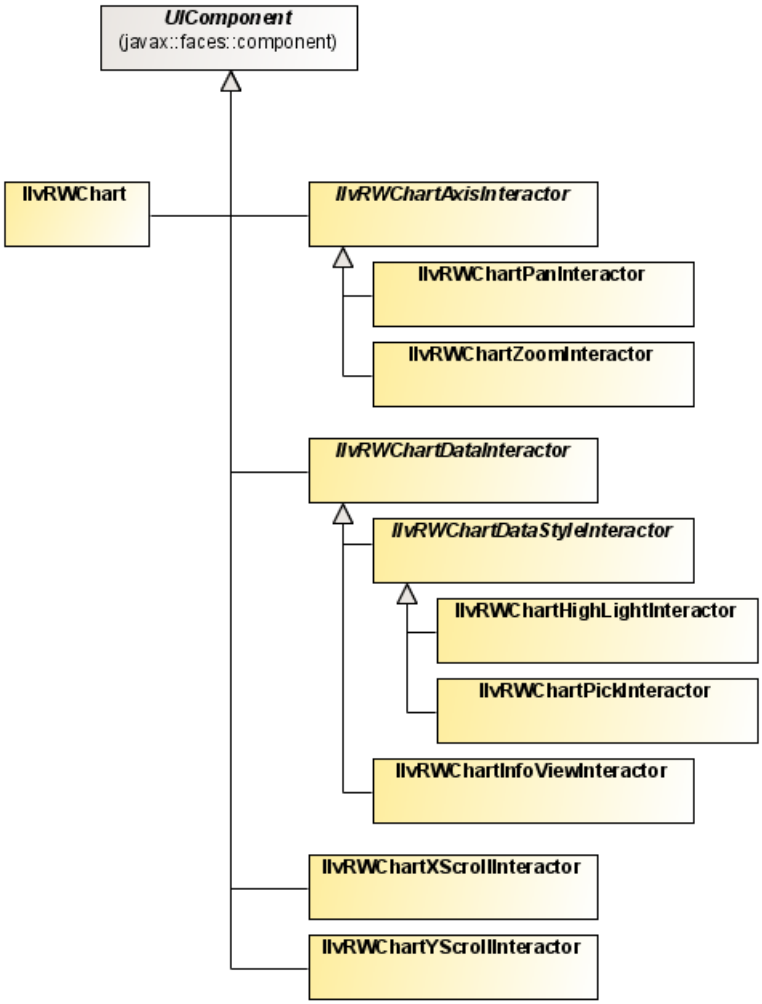
RWChartInteractionEvent
Describes the `RWCChartInteractionEvent` class.

Data source
Describes the server-side data source for a chart.

JSF UI components

The following table lists the JSF tags and the corresponding JSF UI Components.

JSF tag	UI component
chart	IlvRWChart
highlightInteractor	IlvRWChartHighLightInteractor
infoViewInteractor	IlvRWChartInfoViewInteractor
panInteractor	IlvRWChartPanInteractor
pickInteractor	IlvRWChartPickInteractor
xScrollInteractor	IlvRWChartXScrollInteractor
yScrollInteractor	IlvRWChartYScrollInteractor
zoomInteractor	IlvRWChartZoomInteractor



JSF UI components relationships

RWChartInteractionEvent

`RWChartInteractionEvent` is a subclass of `javax.faces.event.ActionEvent`. It provides the data set point involved in the interaction.

This event is delivered to an `ActionListener` or a `RWChartInteractionListener` registered with a `pickInteractor` tag.

For an example of how to use such an event, see *Trigger a server-side action when picking a data point*.

Data source

The server-side data source for a chart may be any instance of `IlvDataSource`. It may be created automatically from the information contained in a project file of the Designer, or you may instantiate it in a managed bean and reference it on the chart tag.

You will find more information on data sources in:

- Data model in *Introducing JViews Charts*
- Using the Data Model in *Developing with the SDK*

For an example of how to retrieve the data source into a `RWChartInteractionListener`, see *Trigger a server-side action when picking a data point*.

Server configuration

In this section you will learn how to configure the Rich Web Charts server.

In this section

Configuring the servlets

In this section you will learn how to configure the servlets required for the execution of a Web application using Rich Web Charts.

Configuring the data source replication

In a Rich Web Charts application the chart data source is replicated from the server to the clients. In this section, you will learn what the replication mechanism consists of, how to configure it, and how its configuration relates to the update interval of the client.

Configuring the servlets

In this section you will learn how to configure the servlets required for the execution of a Web application using Rich Web Charts.

In this section

Introduction
Describes the requirements to configure the servlets required for the execution of a Web application using Rich Web Charts.

Cache servlet
Describes the steps to configure the cache servlet.

Data servlet
Describes how to configure the data servlet.

Resource servlet
Describes how to configure the resource servlet.

Introduction

In this section you will learn how to configure the servlets required for the execution of a Web application using Rich Web Charts. These settings are specified in the file `web.xml` which is the deployment descriptor for the Web application.

For a complete example, see **<installdir>/jviews-charts86/samples/rwc-chart-stock/web.xml**.

The following servlets are required:

- JavaServer Faces implementation servlet
- Rich Web Charts cache servlet
- Rich Web Charts data servlet
- Rich Web Charts resource servlet

The configuration of the JavaServer Faces servlet is not covered in this document. Please consult the documentation for the implementation you are currently using.

In addition to the servlets, the following initialization parameters must be set in the Web deployment descriptor:

Name	Mandatory	Description
`ilog.views.chart.rwc.CACHE_SERVLET_MAPPING`	Yes	The mapping path for the cache servlet.
`log.views.chart.rwc.DATA_SERVLET_MAPPING`	Yes	The mapping path for the data servlet.
`ilog.views.chart.rwc.RESOURCE_SERVLET_MAPPING`	Yes	The mapping path for the resource servlet.

Cache servlet

To configure the cache servlet, go through the following steps:

1. Declare the servlet.

```
<servlet>
  <servlet-name>Cache Servlet</servlet-name>
  <servlet-class>
    ilog.views.chart.rwc.faces.internal.servlet.IlvRWCacheServlet
  </servlet-class>
  <load-on-startup> 1 </load-on-startup>
</servlet>
```

2. Define the URL mapping for the servlet.

   The URL-pattern must be reflected in the mapping path for the servlet, as described in the next step.

```
<servlet-mapping>
  <servlet-name>Cache Servlet</servlet-name>
  <url-pattern>/cache/*</url-pattern>
</servlet-mapping>
```

3. Set the mapping path for the servlet.

   The servlet mapping path must match the URL mapping defined for the servlet. For example, if the URL mapping is `/cache/*`, you must set the corresponding servlet mapping to: `cache`.

   Example:

```
<context-param>
  <param-name>ilog.views.chart.rwc.CACHE_SERVLET_MAPPING</param-name>
  <param-value>cache</param-value>
</context-param>
```

Data servlet

To configure the data servlet, follow the same procedure as described in *Cache servlet*.

Example:

```
<servlet>
  <servlet-name>Data Servlet</servlet-name>
  <servlet-class>
    ilog.views.chart.rwc.data.IlvDataServlet
  </servlet-class>
  <load-on-startup> 1 </load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Data Servlet</servlet-name>
  <url-pattern>/data/*</url-pattern>
</servlet-mapping>

<context-param>
  <param-name>ilog.views.chart.rwc.DATA_SERVLET_MAPPING</param-name>
  <param-value>data</param-value>
</context-param>
```

Resource servlet

To configure the data servlet, follow the same procedure as described in *Cache servlet*.

Example:

```
<servlet>
  <servlet-name>Resource Servlet</servlet-name>
  <servlet-class>
    ilog.views.chart.rwc.resource.IlvResourceServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Resource Servlet</servlet-name>
  <url-pattern>/resource/*</url-pattern>
</servlet-mapping>

<context-param>
  <param-name>
    ilog.views.chart.rwc.RESOURCE_SERVLET_MAPPING
  </param-name>
  <param-value>resource</param-value>
</context-param>
```

Configuring the data source replication

In a Rich Web Charts application the chart data source is replicated from the server to the clients. In this section, you will learn what the replication mechanism consists of, how to configure it, and how its configuration relates to the update interval of the client.

In this section

Purging the history of events
Identifies the conditions under which the history of events in a data source is purged.

Setting the parameters of the data source replication
Describes different scenarios for adjusting the parameters of the data source replication.

Purging the history of events

For each Rich Web Charts component used in an application, the server maintains a history of the events that occurred in the data source associated with the chart component. This history is used to provide the data changes that occurred between two client updates:

- The first time the client connects to the server it retrieves the content from the data source.
- Each time it connects afterwards, the client retrieves the changes that occurred in the data source since the last connection.

To prevent the history from consuming excessive server resources, a purge mechanism is made available. You can control the purge mechanism by using the following parameters:

Name	Mandatory	Description	Default value
`ilog.views.chart.rwc.EVENT_AGE_THRESHOLD`	No	Defines a purge condition, that is checked to true when the history contains events older than this limit. The value is in milliseconds.	600000ms, that is: 10 minutes
`ilog.views.chart.rwc.EVENT_COUNT_THRESHOLD`	No	Defines a purge condition, that is checked to true when the number of events in the history exceeds this limit.	2000
`ilog.views.chart.rwc.PURGE_INTERVAL`	No	The minimum amount of time between successive verifications of the purge conditions. The value is in milliseconds.	60000ms, that is: 1 minute

The purge is performed whenever one of the above conditions is `true`. The purge removes the oldest events that exceed the most constraining threshold.

Setting the parameters of the data source replication

The parameters of the data source replication are Web application initialization parameters that are defined in the `web.xml` file which is the Web deployment descriptor for the Web application.

These parameters must be adjusted according to the client update interval and the rate of updates of the data source.

You are going to see how to set the parameters of the data source replication in three different cases:

Case 1: General

In the general case, proceed as follows:

1. Specify the update interval of the clients: `updateInterval`. This is the same value that you use for the `updateInterval` attribute of the JSF tag `chart`.

   To ensure that the server holds enough data, the history must store at least the changes for twice the duration of the update interval:

2. The event age threshold must be at least 2 * `updateInterval`. You may want to choose higher values to be more tolerant with the client ability to connect to the server.

3. Determine the maximum number of updates of the data source that could occur during `2*updateInterval`: this is the minimum value for the event count threshold.

   The purge interval allows you to control how much the history may exceed the thresholds.

4. Set the purge interval to 20% of the update interval. This value allows the history to exceed the thresholds by 10%.

Example:

For an application that uses a single chart, with an update interval of 2 minutes, and a data source that can have a maximum of 100 updates per minute, you should use the following parameter values in the Web deployment descriptor:

```
[...]
  <context-param>
    <!-- The event age threshold should allow for twice
         the update interval: 2*2 minutes.
         That is: 240000 ms. -->
    <param-name>ilog.views.chart.rwc. EVENT_AGE_THRESHOLD</param-name>
    <param-value>240000</param-value>
  </context-param>
  <context-param>
    <!-- The event count threshold should hold the maximum number
         of updates during twice the update interval:
         (2*2 minutes) * (100 updates/minute) = 400. -->
    <param-name>ilog.views.chart.rwc.EVENT_COUNT_THRESHOLD</param-name>
    <param-value>400</param-value>
  </context-param>
```

```
  <context-param>
    <!-- A purge interval that allows for 10% of threshold excess:
         (2*2 minutes) * 0.10 = 0.4 minutes.
         That is: 24000 ms. -->
    <param-name>ilog.views.chart.rwc.PURGE_INTERVAL</param-name>
    <param-value>24000</param-value>
  </context-param>
 [...]
```

Case 2: Web application with several chart components

The parameters of data source replication are shared by all chart components of a single Web application. Hence, when using several charts in a single Web application, you should use:

- the largest update interval of all charts,
- the largest number of updates of all data sources,

when performing the calculation of the parameters of the data source replication using the steps listed in Case 1.

Case 3: Web application with cyclic data sets

When using a data source with a cyclic data set, each data point you add provokes a complete update of the data set. It is more efficient to send the latest content of the data set to the client, instead of sending each modification. To do so, you can disable the history by setting all parameters to zero.

Example:

```
[...]
  <!-- Disable the event history. -->
  <context-param>
    <param-name>ilog.views.chart.rwc.EVENT_AGE_THRESHOLD</param-name>
    <param-value>0</param-value>
  </context-param>
  <context-param>
    <param-name>ilog.views.chart.rwc.EVENT_COUNT_THRESHOLD</param-name>
    <param-value>0</param-value>
  </context-param>
  <context-param>
    <param-name>ilog.views.chart.rwc.PURGE_INTERVAL</param-name>
    <param-value>0</param-value>
  </context-param>
 [...]
```

Styling

Describes the use of cascaded style sheets to control the appearance of the Rich Web Charts.

In this section

Introduction
Introduces the use of cascaded style sheets to control the appearance of the Rich Web Charts.

Styling the Chart component
Describes the supported CSS properties used to configure the appearance of the Rich Web Charts component.

Styling the data series
Describes the supported CSS properties used to configure the appearance of data series in the Rich Web Charts component.

Property values
Describes the level of support for the following types of objects which can be used as a CSS property value: data indicator, numerical graduation type, paint, stroke, zoom interactor.

Unsupported CSS features
Lists the CSS features that are not supported.

Identifying styling issues
Describes what to do if the chart is not as you expected.

Introduction

The appearance of the Rich Web Charts can be controlled with cascaded style sheets (CSS).

To create a cascaded style sheet for the Rich Web Charts, you can either:

- re-use an existing style sheet created for the IBM® ILOG® JViews Charts component,

or

- author new style sheets using the Designer. See Using the Designer.

Most of the features from the IBM® ILOG® JViews Charts cascaded style sheets are supported by the Rich Web Charts. However, a few properties and property values are not supported. The following sections describe in detail the elements that are in the IBM® ILOG® JViews Charts cascaded style sheet and are supported by the Rich Web Charts:

- *Styling the Chart component*
- *Styling the data series*
- *Property values*
- *Unsupported CSS features*
- *Identifying styling issues*

For information on how to reference a style sheet into a Rich Web Charts component see *Add a Rich Web Charts to a JSP page*.

Styling the Chart component

This section describes the supported CSS properties used to configure the appearance of the Rich Web Charts component. The properties are arranged according to the CSS selectors to which they can be applied. For each selector you will find two tables, as follows:

- The first table lists:
  - the selector as it appears in a CSS,
  - the name of the corresponding style rule as it appears in the English-localized Designer.
- The second table lists:
  - the name of the property as it appears in a CSS file and in the Styling Properties view of the Designer,
  - the location and property name as it appears in the Styling Customizer of the English-localized Designer,
  - whether the property is supported or not.

For a description of the selectors, the properties and their effect, see Styling the Chart Component in Developing with the SDK.

Chart general properties

Selector for the Chart general properties

CSS selector	Name of the style rule in the Designer
chart	Options > Chart General Properties

Chart general properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`3d`	3D View > 3D	No
`antiAliasing`	Appearance > Frame > Anti-aliasing	Yes
`antiAliasingText`	Appearance > Text > Anti-aliasing	Yes
`background`	Appearance > Frame > Background	Yes
`backgroundPaint`	Appearance > Frame > Background	Yes. See also *Paint*
`border`	Appearance > Frame > Border	No
`dataSource`	Not applicable	No
`decorations`	Menu Chart/Image Decoration	Yes

CSS property	Property name in the Styling Customizer of the Designer	Support
		Note: Instances of `IlvDataIndicator` are supported. Instances of `IlvThresholdIndicator` are not supported. See *Data indicator* for details on the level of support for `IlvDataIndicator`.
`defaultColors`	Not applicable	No
`dynamicStyling`	Not applicable	No
`font`	Appearance > Text > Font	Yes
`footer`	Not applicable	No
`footerText`	Footer > Text > Text	No
`foreground`	Appearance > Frame > Foreground	Yes
`header`	Not applicable	No
`headerText`	Header > Text > Text	No
`interactors`	Interactions > Interactors	Yes **Note**: Instances of `IlvChartHighlightInteractor`, `IlvChartInfoViewInteractor`, `IlvChartPanInteractor`, `IlvChartPickInteractor`, `IlvChartXScrollInteractor`, `IlvChartYScrollInteractor`, `IlvChartZoomInteractor` are supported. In the Designer these correspond to: Highlight, InfoView, Pan, Pick, XScroll, YScroll and Zoom. See *Zoom interactor* for details on the level of support for `IlvChartZoomInteractor`.
`legendPosition`	Position	Yes

CSS property	Property name in the Styling Customizer of the Designer	Support
	Note: The property is located in the Styling Customizer of the Designer when the option Legend is selected in the Style Rules tree pane.	**Note**: The following CSS property values are supported: North_Bottom, North_West, West, South_West, South_Top, South_East, East, North_East. In the Designer these correspond to all Docked values.
`legendVisible`	Not applicable	Yes
`opaque`	Appearance > Frame > Opaque	Yes
`optimizedRepaint`	Not applicable	No
`plotAreaBackground`	Appearance > Plotting Area > Plotting Area Background **Note**: The property is located in the Styling Customizer of the Designer when the option Chart Area is selected in the Style Rules tree pane.	Yes
`projectorReversed`	Series Rendering > Projector reversed	Yes
`renderingType`	Series Rendering > Representation	Yes **Note**: The following CSS property values are supported: BAR, STACKED_BAR, STACKED100_BAR, SUPERIMPOSED_BAR, STACKED_POLYLINE, STACKED100_POLYLINE, SUPERIMPOSED_POLYLINE, SCATTER, AREA,

CSS property	Property name in the Styling Customizer of the Designer	Support
		STACKED_AREA, STACKED100_AREA, STAIR, STACKED_STAIR, SUMMED_STAIR, STACKED100_STAIR, BUBBLE, HILO, CANDLE, HLOC, HILO_ARROW, HILO_STICK, PIE, COMBO. In the Designer these correspond to: Bar, Polyline, Scatter, Area, Stair, High-Low and Pie representations, and the Combined representation using the supported representations.
`scalingFont`	Appearance > Text > Scale when resizing	No
`scrollRatio`	Not applicable	Yes
`shiftScroll`	Not applicable	Yes
`type`	Series Rendering > Chart Type	Yes **Note**: The following CSS property values are supported: CARTESIAN and PIE. In the Designer these

CSS property	Property name in the Styling Customizer of the Designer	Support
		correspond to: Cartesian chart and Pie chart types.
`visible`	Not applicable	No

Chart area

Selector for Chart area

CSS selector	Name of the style rule in the Designer
`chartArea`	Options > Chart Area

Chart area properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`background`	Appearance > Colors > Background	Yes
`backgroundPaint`	Appearance > Colors > Background	Yes. See also *Paint*
`border`	Appearance > Colors > Border	No
`bottomMargin`	Not applicable	No
`filledPlottingArea`	Appearance > Plotting Area > Filled	Yes
`font`	Appearance > Colors > Font	Yes
`foreground`	Appearance > Colors > Foreground	Yes
`leftMargin`	Not applicable	No
`margins`	Appearance > Margins	Yes
`opaque`	Appearance > Colors > Opaque	Yes
`plotBackground`	Appearance > Plotting Area > Plotting Area Background	Yes. See also *Paint*
`plotStyle`	Not applicable	Yes
`rightMargin`	Not applicable	No

Grids

Selectors for grids

CSS selector	Name of the style rule in the Designer

CSS selector	Name of the style rule in the Designer
`chartGrid`	Options > Grids
`chartGrid[axisIndex="-1"]`	Options > Grids > X Grid
`chartGrid[axisIndex="0"]`	Options > Grids > Y Grid
`#xGrid`	Options > Grids > X Grid
`#yGrid`	Options > Grids > Y Grid

Grids properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`drawOrder`	Appearance > Visibility > Drawing order	Yes
`majorLineVisible`	Appearance > Major lines > Visible	Yes
`majorPaint`	Appearance > Major lines > Color	Yes. See also *Paint*
`majorStroke`	Appearance > Major lines > Stroke	Yes. See also *Stroke*
`minorLineVisible`	Appearance > Minor lines > Visible	Yes
`minorPaint`	Appearance > Minor lines > Color	Yes. See also *Paint*
`minorStroke`	Appearance > Minor lines > Stroke	Yes. See also *Stroke*
`visible`	Not applicable	Yes

Legend

Selector for legend

CSS selector	Name of the style rule in the Designer
`chartLegend`	Options > Legend

Legend properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`antiAliasing`	Appearance > Symbol > Anti-aliasing	Yes
`antiAliasingText`	Appearance > Text > Anti-aliasing	Yes
`background`	Appearance > Frame > Background	Yes
`border`	Appearance > Frame > Border	No
`floating`	Position > Floating	No
`floatingLayoutDirection`	Position > Floating	No
`followChartResize`	Position > Floating	No

CSS property	Property name in the Styling Customizer of the Designer	Support
`font`	Appearance > Text > Font	Yes
`foreground`	Appearance > Text > Color	Yes
`symbolTextSpacing`	Appearance > Symbol > Spacing	Yes
`interactive`	Not applicable	No
`location`	Not applicable	No
`movable`	Not applicable	No
`paintingBackground`	Appearance > Frame > Opaque	Yes
`symbolSize`	Appearance > Symbol > Dimension	Yes
`title`	Not applicable	Yes
`transparency`	Appearance > Frame > Background	Yes

Scales

Selectors for scales

CSS selector	Name of the style rule in the Designer
`chartScale`	Options > Scales
`chartScale[axisIndex="-1"]`	Options > Scales > X Scale
`chartScale[axisIndex="0"]`	Options > Scales > Y Scale
`#xScale`	Options > Scales > X Scale
`#yScale`	Options > Scales > Y Scale

Scales properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`annotations`	Not applicable	No
`autoCrossing`	Scale > Appearance > Auto-crossing	Yes
`autoWrapping`	Labels > Multiline labels > Auto wrapping	No
`axisStroke`	Scale > Appearance > Axis stroke	Yes. See also *Stroke*
`axisVisible`	Scale > Appearance > Axis visible	Yes

CSS property	Property name in the Styling Customizer of the Designer	Support
`category`		No
`crossingValue`	Scale > Appearance > Auto-crossing > Value	Yes
`drawOrder`	Not applicable	Yes
`foreground`	Scale > Appearance > Color	Yes
`labelAlignment`	Labels > Multiline labels > Label alignment	No
`labelColor`	Labels > Appearance > Color	Yes
`labelFont`	Labels > Appearance > Font	Yes
`labelOffset`	Labels > Appearance > Offset	Yes
`labelRotation`	Labels > Appearance > Rotation	Yes
`labelVisible`	Labels > Appearance > Show scale labels	Yes
`majorTickSize`	Scale > Ticks > Major ticks size	Yes
`majorTickVisible`	Scale > Ticks > Show major ticks	Yes
`minorTickSize`	Scale > Ticks > Minor ticks size	Yes
`minorTickVisible`	Scale > Ticks > Show minor ticks	Yes
`skipLabelMode`	Labels > Overlapping rules > Skipping mode	Yes
`skippingLabel`	Labels > Overlapping rules > Skip overlapping labels	Yes
`stepsDefinition`	Axis > Graduations	Yes **Note**: Instances of `IlvCategoryStepsDefinition`, `IlvTimeStepsDefinition`, `IlvDefaultStepsDefinition` are supported. The corresponding

CSS property	Property name in the Styling Customizer of the Designer	Support
		graduation types in the Designer are: Category, Time, Numerical. See *Numerical graduation type* for details

CSS property	Property name in the Styling Customizer of the Designer	Support
		on the support level of `IlvDefaultStepsDefinition`.
`tickLayout`	Not applicable	No
`title`	Titles > Appearance > Title	Yes
`titleOffset`	Titles > Appearance > Offset	Yes
`titlePlacement`	Titles > Appearance > Placement	Yes
`titleRotation`	Titles > Appearance > Rotation	Yes
`visible`	Scale > Appearance > Visible	Yes

Axis

Selectors for axis

CSS selector	Name of the style rule in the Designer
`chartScale axis`	Options > Scales
`chartScale[axisIndex="-1"] axis`	Options > Scales > X Scale
`chartScale[axisIndex="0"] axis`	Options > Scales > Y Scale
`#xScale axis`	Options > Scales > X Scale
`#yScale axis`	Options > Scales > Y Scale

Axis properties

CSS property	Property name in Styling Customizer of the Designer	Support
`autoDataRange`	Axis > Ranges > Automatic Data Range	Yes
`autoVisibleRange`	Axis > Ranges > Automatic Visible Range	Yes
`dataMin`	Axis > Ranges > Automatic Data Range > Min	Yes
`dataMax`	Axis > Ranges > Automatic Data Range > Max	Yes
`reversed`	Axis > Ranges > Reverse Axis	Yes
`visibleMin`	Axis > Ranges > Automatic Visible Range > Min	Yes

CSS property	Property name in Styling Customizer of the Designer	Support
`visibleMax`	Axis > Ranges > Automatic Visible Range > Min	Yes
`Transformer`	Axis > Graduations > Transformer	No

Styling the data series

This section describes the supported CSS properties used to configure the appearance of data series in the Rich Web Charts component. The properties are arranged according to the CSS selectors to which they can be applied. For each selector you will find two tables, as follows:

- The first table lists:
  - the CSS type and CSS attributes usable in the selectors, as they appear in a CSS,
  - the name of the corresponding style rule as it appears in the English-localized Designer.
- The following tables list:
  - the name of the property as it appears in a CSS file and in the Styling Properties view of the Designer,
  - the location and property name as it appears in the Styling Customizer of the English-localized Designer,
  - whether the property is supported or not.

For a description of the selectors, the properties and their effect, see Styling the data series in Developing with the SDK.

Series

Selector components for series

CSS type	CSS attributes	Name of the style rule in the Designer
series	name, index	series

Series properties available for all renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`labeling`	Data Labels > Default Labeling > Mode	No
`labelLayout`	Data Labels > Default Labeling > Layout	No
`visible`	Series > Appearance > Visible	Yes
`visibleInLegend`	Series > Appearance > Visible in Legend	Yes
`annotation`	Not applicable	No
`color1`	Series > Appearance > Primary color	Yes. See also *Paint*
`color2`	Series > Appearance > Secondary color	Yes. See also *Paint*
`endcap`	Series > Appearance > Line/Contour Stroke > End Cap	Yes
`lineJoin`	Series > Appearance > Line/Contour Stroke > Line Join	Yes
`lineStyle`	Series > Appearance > Line/Contour Stroke > Line Style	Yes
`lineWidth`	Series > Appearance > Line/Contour Stroke > Line Width	Yes
`miterLimit`	Not applicable	Yes
`renderingType`	Series Rendering > Appearance > Representation **Note**: The property is located in the Styling Customizer of the Designer when the option Chart General Properties is selected in the Style Rules tree pane.	Yes **Note**: The following CSS property values are supported: BAR, STACKED_BAR, STACKED100_BAR, SUPERIMPOSED_BAR, STACKED_POLYLINE, STACKED100_POLYLINE, SUPERIMPOSED_POLYLINE, SCATTER, AREA, STACKED_AREA,

CSS property	Property name in the Styling Customizer of the Designer	Support
		STACKED100_AREA, STAIR, STACKED_STAIR, SUMMED_STAIR, STACKED100_STAIR, BUBBLE, HILO, CANDLE, HLOC, HILO_ARROW, HILO_STICK. In the Designer these correspond to: Bar,

CSS property	Property name in the Styling Customizer of the Designer	Support
		Polyline, Scatter, Area, Stair and High-Low representations.
`stroke`	Series > Appearance > Line/Contour Stroke	Yes. See also *Stroke*
`style`	Not applicable	Yes

Series properties available for bar renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`autoTransparency`	Not applicable	No. `autotransparency` is always on
`overlap`	Not applicable	No
`widthPercent`	Not applicable	No

Series properties available for bubble renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`marker`	Series > Marker > Type	Yes
`minSize`	Series > Bubbles	Yes
`maxSize`	Series > Bubbles	Yes

Series properties available for high-low renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`overlap`	Not applicable	No
`widthPercent`	Not applicable	No

Series properties available for pie chart renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`holeSize`	Not applicable	Yes
`strokeOn`	Series > Appearance > Line/Contour Stroke	Yes

Series properties available for polyline renderers

CSS property	Property name in the Styling Customizer of the Designer	Support

CSS property	Property name in the Styling Customizer of the Designer	Support
`autoTransparency`	Not applicable	No. `autotransparency` is always on
`marker`	Series > Marker > Type	Yes
`markerSize`	Series > Marker > Size	Yes

Series properties available for scatter chart renderers

CSS property	Property name in the Styling Customizer of the Designer	Support
`marker`	Series > Marker > Type	Yes
`markerSize`	Series > Marker > Size	Yes

Point

Selector components for point

CSS type	CSS attributes	Name of the style rule in the Designer
`point`	index, label, x, y, seriesIndex, seriesName	point
`series > point`	For series: index, name	
`For point: index, label, x, y`	Not applicable	

Point properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`annotation`	Not applicable	No
`color1`	Data Points > Appearance > Primary color	Yes. See also *Paint*
`color2`	Data Points > Appearance > Secondary color	Yes. See also *Paint*
`endcap`	Data Points > Appearance > Line/Contour Stroke > End Cap	Yes
`labeling`	Data Labels > Default Labeling > Mode	No
`labelLayout`	Data Labels > Default Labeling > Layout	No
`lineJoin`	Data Points > Appearance > Line/Contour Stroke > Line Join	Yes
`lineStyle`	Data Points > Appearance > Line/Contour Stroke > Line Style	Yes
`lineWidth`	Data Points > Appearance > Line/Contour Stroke > Line Width	Yes
`marker`	Data Points > Marker > Type	Yes

CSS property	Property name in the Styling Customizer of the Designer	Support
`miterLimit`	Not applicable	Yes
`stroke`	Data Points > Appearance > Line/Contour Stroke	Yes. See also *Stroke*
`visible`	Data Points > Appearance > Visible	Yes

Property values

This section describes in detail the level of support for the following types of objects which can be used as a CSS property value:

- *Data indicator*
- *Numerical graduation type*
- *Paint*
- *Stroke*
- *Zoom interactor*

For Data Indicator, Numerical Graduation Type and Zoom Interactor, you will find two tables, as follows:

The first table lists:

- the supported values for the class property, as they appear in a CSS
- the location in the Designer where such a value can be created or configured.

The second table lists:

- the name of the property as it appears in a CSS file and in the Styling Properties view of the Designer,
- the location and property name as it appears in the Styling Customizer of the English-localized Designer,
- whether the property is supported or not.

Data indicator

Supported class for data indicator

CSS 'class' property	Location in the Designer

`IlvDataIndicator(int, ilog.views.chart.IlvDataInterval, java.lang.String)`
 	
`IlvDataIndicator(int, ilog.views.chart.IlvDataWindow, java.lang.String)`
 	
`IlvDataIndicator(int, double, java.lang.String)`
	Menu Chart/X-Range, Y-Range, X-Value, Y-Value Data Indicator; Data Window Indicator

Data indicator properties

CSS property	Location in the Designer	Support
`axisIndex`	Not applicable	Yes
`dataWindow`	Data > Domain Definition > X-Range, Y-Range	Yes
`labelRenderer`	Not applicable	No
`range`	Data > Domain Definition > X-Range or Y-Range	Yes
`style`	Appearance > Lines and Colors Appearance > Text	Yes
`text`	Appearance > Text > Label	Yes
`value`	Data > Domain Definition > Value	Yes

Numerical graduation type

Supported class for numerical graduation type

CSS 'class' property	Location in the Designer
`IlvDefaultStepsDefinition`	Axis > Graduations > Numerical Steps Configuration

Numerical graduation type properties

CSS property	Property name in the Styling Customizer of the Designer	Support
`autoMode`	Not applicable	No
`autoNumberFormat`	Not applicable	No
`autoStepUnit`	Automatic Step Unit Calculation	Yes
`autoSubStepUnit`	Automatic Substep Unit Calculation	Yes
`numberFormat`	Not applicable	No
`stepUnit`	Automatic Step Unit Calculation > Step Unit	Yes

CSS property	Property name in the Styling Customizer of the Designer	Support
subStepCount	Not applicable	No
subStepUnit	Automatic Substep Unit Calculation > Steps Unit	Yes

Paint

Properties that accept a `java.awt.Paint` value support the following classes:

- `java.awt.Color`
- `java.awt.GradientPaint`
- ilog.views.util.java2d. `IlvLinearGradientPaint`
- ilog.views.util.java2d. `IlvRadialGradientPaint`

Other classes, including ilog.views.util.java2d. `IlvTexture` and ilog.views.util.java2d. `IlvPattern`, are not supported.

Stroke

Properties that accept a `java.awt.Stroke` value support the following class: `java.awt.BasicStroke`.

Zoom interactor

Supported class for zoom interactor

CSS 'class' property	Location in the Designer
IlvChartZoomInteractor	Interactions > Settings

Zoom interactor properties

CSS property	Property name in the Styling Customizer of the Designer	Support
animationStep	Animation steps	No
XZoomAllowed	Zoom along the x-axis	Yes
YZoomAllowed	Zoom along the y-axis	Yes
zoomOutAllowed	Zoom out allowed	Yes

Unsupported CSS features

The following CSS features are not supported:

- CSS expressions
- CSS functions (with the exception of the @|interactors and @|decorations functions)
- selectors using pseudo-classes
- selectors using CSS classes
- selectors using transitions of type E > F or E + F
- the 'Any element' selector: *
- empty selector

For more details on CSS expressions and functions, see Expressions in Developing with the SDK.

For more details on selectors see Selector in Developing with the SDK.

Identifying styling issues

If the chart is not as you expected, you should do the following:

1. Check the style sheet with the Designer.
   - If the chart does not look like as you expect in the Designer, it will not display properly in the Rich Web Charts.
   - Use the Messages view of the Designer to make sure the style sheet syntax is correct.
2. Check the web application log file for messages regarding styling features.

Index