



IBM ILOG JViews Gantt V8.6

Building Web Applications

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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 Gantt.....	7
Overview.....	8
Thin client applications.....	9
Thin client application designs.....	10
Ajax-enabled components.....	11
Rich Web applications.....	13
Overview.....	14
Applets.....	15
Java Web Start applications.....	16
Using DHTML-based JSF components to build Web applications.....	17
Introduction.....	18
The architecture of JViews Gantt Faces.....	19
About support for JViews Gantt Faces.....	20
Servlet and component classes.....	21
The JViews Gantt Faces component set.....	25
Creating simple views.....	27
JViews Gantt Designer project.....	29
JViews Charts Designer project.....	30
Data source binding in JViews Gantt.....	31
Data source binding in JViews Charts.....	34
Styling Gantt chart data with CSS.....	36

Styling chart data with CSS.....	37
Installing interactors in a Gantt chart.....	38
Installing interactors in a chart.....	39
Select interactor.....	40
Connecting a Gantt chart to a message box.....	48
Connecting a chart view to a message box.....	49
Adding a popup menu.....	50
Styling the popup menu.....	53
Managing the session expiration.....	54
JavaScript objects.....	55
Contexts for actions on the Gantt Chart view.....	57
Introduction.....	58
JavaServer Faces lifecycle context.....	59
Image servlet context.....	62
Integrating JViews Faces in your environment.....	63
JViews Faces configuration at JViews Framework level.....	64
Session persistence.....	66
Running JViews Faces components in JSR 168 portlets.....	67
Guide to using JViews components with ICEfaces.....	71
Settings for using JViews components in ICEfaces.....	72
Interoperability between JViews components and ICEfaces components.....	73
Push updates to JViews components.....	74
ICEfaces software in JViews.....	75
Supporting Facelets and Trinidad.....	76
Web Application Server support.....	77
Deploying an application as a DHTML-only thin client.....	79
JavaServer Faces components as opposed to DHTML thin client.....	81
Overview.....	82
Gantt Thin-Client Web Architecture.....	83
Getting Started With the Gantt Thin Client: An Example.....	85
Creating a Gantt thin-client application.....	86
The Gantt Servlet Example.....	87
Installing and Running the Gantt Servlet Example.....	89
Developing the server side.....	91
Key classes and their associations.....	92
The servlet support class.....	94
Multithreading issues on the server side.....	96
The servlet class.....	97
Answering HTTP requests.....	98
Developing the client side.....	99

Developing a Dynamic HTML client.....	100
The DHTML client for the Gantt Servlet example.....	104
The Popup menu in JavaScript.....	124
Adding client/server interactions.....	127
The client side.....	128
The server side.....	131
Actions that modify chart capabilities.....	132
The IlvGanttServlet and IlvGanttServletSupport classes.....	135
Creating a servlet.....	136
The servlet parameters.....	138
Multiple sessions.....	140
DHTML thin-client support in JViews Framework.....	143
Overview of thin-client support.....	145
IBM® ILOG® JViews thin-client Web architecture.....	146
Getting started with the IBM® ILOG® JViews thin client.....	147
Installing and running the XML Grapher example.....	149
Developing the server.....	150
Developing the client.....	155
Overview of client-side development.....	157
The IlvView JavaScript component.....	158
The IlvOverview JavaScript component.....	161
The IlvLegend JavaScript component.....	163
The IlvButton JavaScript component.....	165
The IlvZoomTool JavaScript component.....	171
The IlvZoomInteractor JavaScript component.....	172
IlvPanInteractor.....	174
The IlvPanTool JavaScript component.....	175
The IlvMapInteractor and IlvMapRectInteractor JavaScript components.....	176
The Popup menu in JavaScript.....	177
Adding client/server interactions.....	180
Generating a client-side image map.....	182
The IlvManagerServlet class.....	185
Overview of the predefined servlet.....	186
The servlet requests and parameters.....	187
Multiple sessions.....	192
Multithreading issues.....	194
The IlvManagerServletSupport class.....	195
Controlling tiling.....	197
Tiling.....	198

Tile size.....	199
Cache mechanisms.....	200
Developing client-side tiling.....	201
Developing server-side tiling.....	203
Client-side caching.....	204
Server-side caching and the tile manager.....	205
Index.....	207

Introducing the Web technologies used in JViews Gantt

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 Gantt 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 Gantt 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 Gantt 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.

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 Gantt helps you to develop such applications as:

- ◆ Applets
- ◆ Java™ Web Start Applications

Applets

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

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

When developing a JViews Gantt application using this approach, see [Developing with the JViews Gantt 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 Gantt application using this approach, see *Developing with the JViews Gantt 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

Shows you how to use the components of JViews Gantt Faces to create JavaServer Pages (JSP) that are compliant with JavaServer Faces (JSF)

In this section

Introduction

Introduces JViews Gantt Faces.

The architecture of JViews Gantt Faces

Presents an overview of the architecture of JViews Gantt Faces.

The JViews Gantt Faces component set

Presents some examples to illustrate how to use JViews Gantt Faces components.

JavaScript objects

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

Contexts for actions on the Gantt Chart 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

This section shows you how to use the components of IBM® ILOG® JViews Gantt Faces to create JavaServer™ Pages (JSP) that are compliant with JavaServer Faces (JSF). JViews Gantt 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 Gantt Faces provide Ajax-enabled components for developing highly responsive and interactive Web applications.

The source code for the examples described in this section is provided under the directory:

```
<installdir>/jviews-gantt86/codefragments/jsf
```

The architecture of JViews Gantt Faces

Presents an overview of the architecture of JViews Gantt Faces.

In this section

About support for JViews Gantt Faces

Describes thin-client support based on JavaServer Faces (JSF) technology.

Servlet and component classes

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

About support for JViews Gantt Faces

JViews Gantt 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 Gantt Faces component set includes:

- ◆ A Gantt chart view.
- ◆ A Schedule chart view.
- ◆ A set of interactors.
- ◆ A popup menu

Not all the components have a visual representation. For example, an interactor 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 a server roundtrip.

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

Servlet and component classes

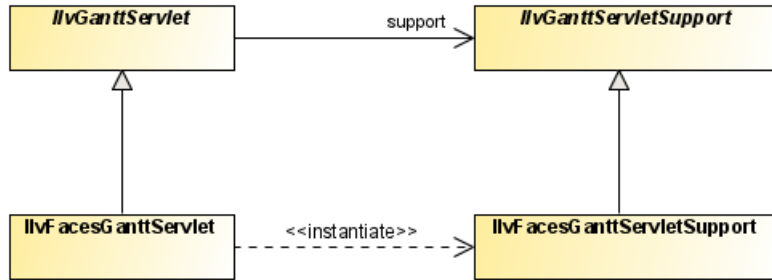
JSF Components in JViews Gantt use servlet technology to produce the images that are the visual representation of the component on the client side. Dedicated servlet, servlet support, and components are available to help create an application.

Servlet and component classes

Name	Description
In package <code>ilog.views.gantt.faces.dhtml.servlet.IlvFacesGanttServlet</code>	A dedicated <code>IlvGanttServlet</code> .
In package <code>ilog.views.gantt.faces.dhtml.servlet.IlvFacesGanttServletSupport</code>	A dedicated <code>IlvGanttServletSupport</code> .
In package <code>ilog.views.gantt.faces.dhtml.component.IlvFacesDHTMLGanttChartView</code>	A view component extended to have DHTML rendering that displays an <code>IlvGanttChart</code> .
In package <code>ilog.views.gantt.faces.dhtml.component.IlvFacesDHTMLScheduleChartView</code>	A view component extended to have DHTML rendering that displays an <code>IlvScheduleChart</code> .
In package <code>ilog.views.gantt.faces.interactor.IlvFacesRowExpandCollapseInteractor</code>	An interactor that allows a hierarchy node (activity or resource) to be expanded or collapsed.
In package <code>ilog.views.gantt.faces.interactor.IlvFacesSheetScrollInteractor</code>	An interactor that allows you to pan the sheet view.
In package <code>ilog.views.gantt.faces.interactor.IlvFacesTableScrollInteractor</code>	An interactor that allows you to pan the table view.
In package <code>ilog.views.gantt.faces.interactor.IlvFacesRowSelectInteractor</code>	An interactor that allows you to execute an action in the servlet context by clicking the image.
In package <code>ilog.views.gantt.faces.dhtml.interactor.IlvFacesNodeSelectInteractor</code>	An interactor that allows you to execute an action in the JSF context by clicking the image.
In package <code>ilog.views.faces.component.IlvFacesContextualMenu</code>	A contextual popup menu.
In package <code>ilog.views.gantt.faces.dhtml.interactor.IlvFacesGanttSelectInteractor</code>	An interactor that allows you to select activities, reservations, and constraints and move activities and reservations in the JSF context.
In package <code>ilog.views.gantt.faces.dhtml.component.IlvFacesGanttSelectionManager</code>	A component that allows you to configure the management of the selection on an <code>IlvFacesDHTMLGanttChartView</code> or <code>IlvFacesDHTMLScheduleChartView</code> component.

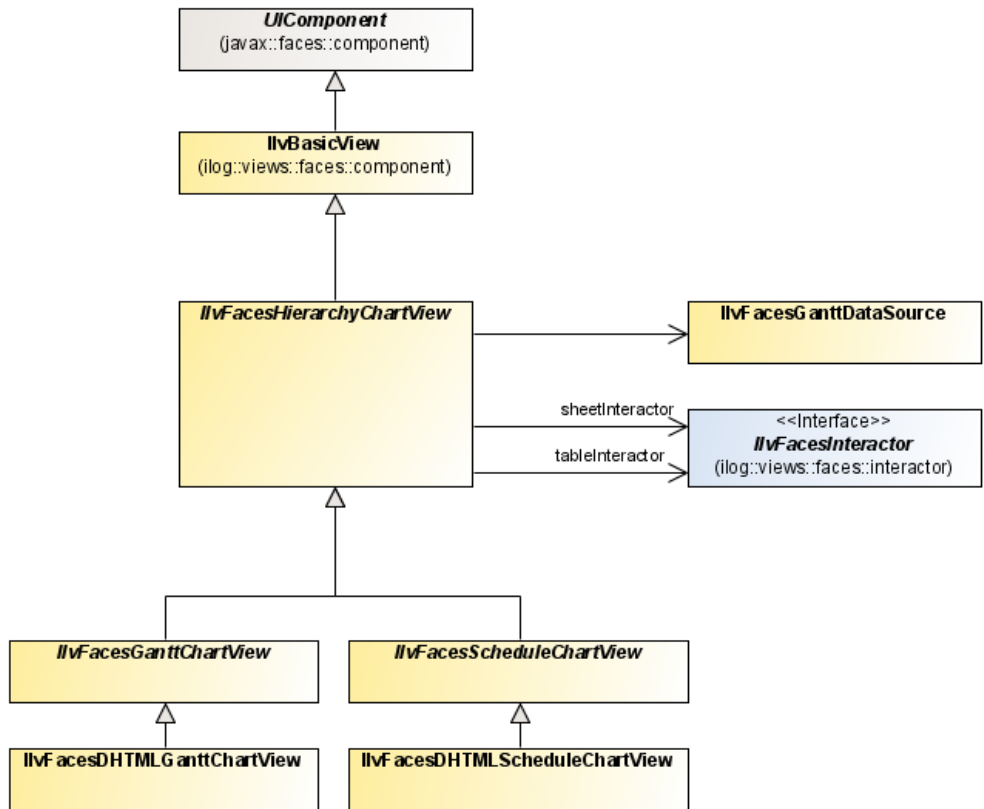
The associations between these classes are shown graphically in the following figures.

Servlet classes shows the servlet classes and their supporting classes.



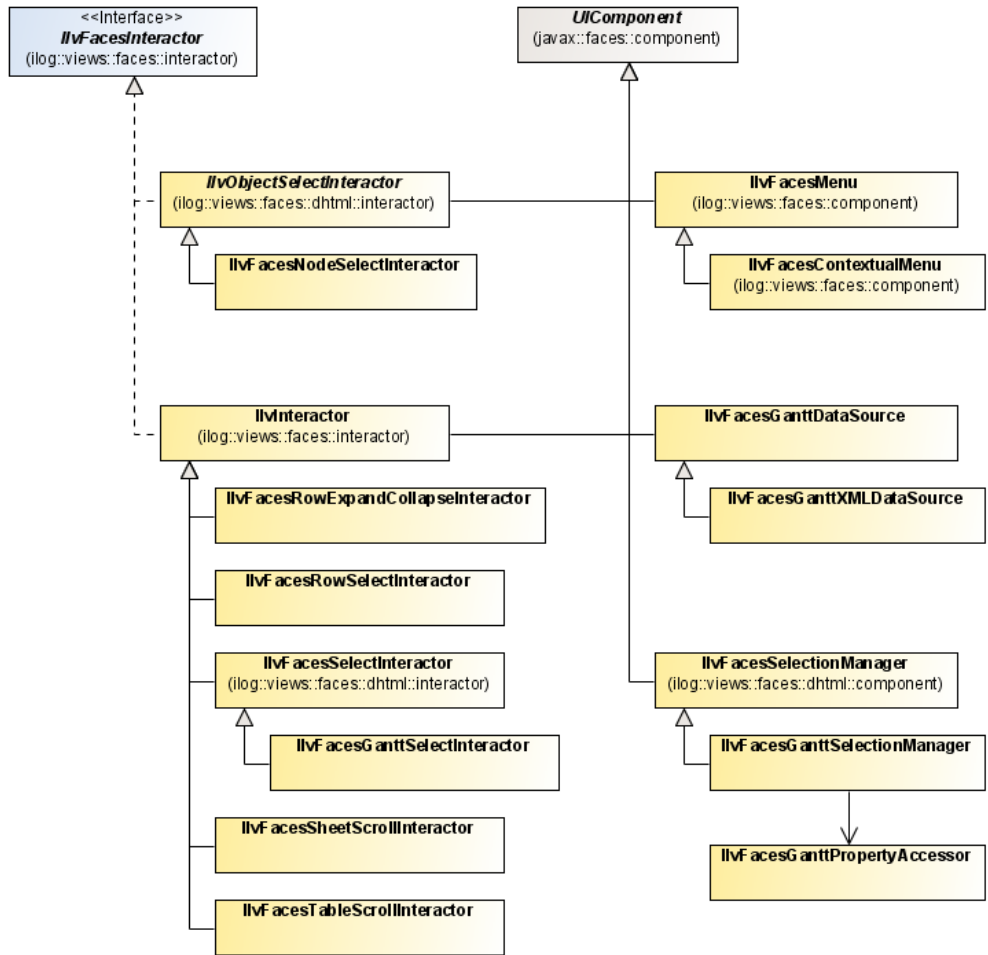
Servlet classes

Base UI component classes shows the Gantt chart view components and their associated classes.



Base UI component classes

The following figure shows the interactor, popup menu, and data source components.



Interactor, popup menu, and data source components

The JViews Gantt Faces component set

Presents some examples to illustrate how to use JViews Gantt Faces components.

In this section

Creating simple views

Explains how to create various types of simple view.

JViews Gantt Designer project

Presents an example using the Designer for JViews Gantt.

JViews Charts Designer project

Presents an example using the Designer for JViews Charts.

Data source binding in JViews Gantt

Presents examples of connecting data source components to Gantt chart components.

Data source binding in JViews Charts

Presents examples of connecting data source components to chart components.

Styling Gantt chart data with CSS

Describes how to customize data display in a Gantt chart by using Cascading Style Sheets.

Styling chart data with CSS

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

Installing interactors in a Gantt chart

Describes how to install interactors in a Gantt chart.

Installing interactors in a chart

Describes how to install interactors in a chart.

Select interactor

Describes the use and behavior of the select interactor in a Gantt chart..

Connecting a Gantt chart to a message box

Describes how to connect a message box to a Gantt chart view.

Connecting a chart view to a message box

Describes how to connect a message box to a `chartView`.

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 Gantt chart view

Creating an empty view

```
<jvgf:ganttView style="width:500px; height:300px;" />
```

This produces a 500 by 300 pixel Gantt chart.

Declaring the namespace

The namespace `jvgf` (for JViews Gantt) must be declared in the page as follows:

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

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.

```
<jvgf:ganttView width="500" height="500" />
```

Note: The Schedule chart view component is used in the same way (tag attributes, behavior, and so on) as the Gantt chart view component. The fundamental difference concerns the wrapped component that is used to produce a view: an instance of `IlvScheduleChart` or of `IlvGanttChart`. Other practical differences are mentioned where appropriate.

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" />
```

JViews Gantt Designer project

The easiest way to configure the style and the data source of a hierarchy chart (Gantt or Schedule chart) is to set a JViews Gantt Designer project to the Gantt view component. This is done with the `data` attribute of the tag that points to an `igpr` file.

For more information about the Designer for JViews Gantt, see [Using the Designer](#).

```
<jvgf:ganttView id="gantt"  
    data="data/gantt.igpr"  
    style="width:800;height:400" />
```

You can set the CSS stylesheet and data source separately.

JViews 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 Gantt

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

Using an SDXL file

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

```
<jvgif:ganttView style="width:500 px; height:300 px;"
  data="resources/gantt.sdxl" />
```

Using a value binding

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

```
<jvgif:ganttView [...] data="#{ganttBean.ganttModel}" />
```

The bean should then provide the Gantt data model through its `getGanttModel` method:

```
public IlvGanttModel getGanttModel() {
    if (ganttModel == null)
        ganttModel = createDefaultModel();
    return ganttModel;
}
```

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 gantt demo bean</description>
    <managed-bean-name>ganttBean</managed-bean-name>
    <managed-bean-class>GanttBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

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

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

DataSource and XMLDataSource components

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

Setting a Data Source on a Gantt Chart Component

```
<jvgf:XMLDataSource filename="resources/gantt.sdxl
                    id="xmlDataSource" />

<jvgf:dataSource value="#{ganttBean.ganttModel}" />

<jvgf:ganttView id="gantt" data="xmlDataSource" [...] />

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

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

In this example, we create two data sources: one filled from an XML file and another from a bound Gantt data model.

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

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

See *JavaScript objects* to learn how to use these proxies.

Creating a component in a managed bean

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

```
<jvgf:ganttView id="chart"
               style="width:500;height:300;"
               chart="#{ganttBean.gantt}" />
```

Here the `IlvGanttChart` 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 IlvGanttChart getGantt() {
    if (ganttChart == null) {
        ganttChart = new IlvGanttChart();
        ganttChart.setGanttModel(getGanttModel());
    }
}
```



```
    }  
    return ganttChart;  
}
```

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

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.

```
<jvpcf: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:

```
<jvpcf: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 Gantt 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:

```
<jvgf:ganttView id="gantt" [...] styleSheets="data/gantt.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.

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.

Installing interactors in a Gantt chart

You can now install interactors in the Gantt chart to interact with it. For example, install a scroll interactor in the table and sheet views:

```
<jvgf:sheetScrollInteractor id="sheetScroll" />
<jvgf:tableScrollInteractor id="tableScroll" />
<jvgf:ganttView tableInteractorId="tableScroll"
  sheetInteractorId="sheetScroll"/>
```

Different interactors can be installed in the table and the sheet views. For example, you can install a row expand or collapse interactor on the table view and a scroll interactor in the sheet view.

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.

Select interactor

The select interactor allows you to:

- ◆ select activities, resources, reservations or constraints, either in the Gantt table or in the Gantt sheet,
- ◆ move activities and reservations in the Gantt sheet.

These operations occur without performing a full refresh of the page.

To define select interactors for the Gantt table and Gantt sheet use the following tags:

```
<jvopf:selectInteractor id="tableSelect">
<jvopf:selectInteractor id="sheetSelect">
```

To set them on a view, use the following code:

```
<jvopf:ganttView id="thegantt"
  [...]
  tableInteractorId="tableSelect"
  sheetInteractorId="sheetSelect">
```

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

Moving the selected items

The select interactor allows you to move the selection when:

- ◆ The `moveAllowed` property of the interactor is set to `true` (default value).
- ◆ The server-side object is movable. By default, activities and reservations are movable. Resources and constraints cannot be moved.

Configuring the rendering of the selection

The way the selection is performed and displayed can be customized on the `ganttView` tag by using a selection manager facet as follows:

```
<jvopf:ganttView id="thegantt"
  tableInteractorId="tableSelect"
  sheetInteractorId="sheetSelect">
  <facet name="tableSelectionManager">
    <jvopf:selectionManager imageMode="false" [...] />
  </facet>
```



```
<facet name="sheetSelectionManager">
  <jvgf:selectionManager imageMode="false" [...] />
</facet>
</jvgf:ganttView>
```

The selection manager has two display modes:

◆ **Image mode (default)**

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

◆ **Regular mode**

Rectangles representing the selection are displayed on top of the view. The roundtrip to the server is minimal; it does not require a new image to be generated and therefore the response time is shorter but the selection feedback is limited to a selection rectangle.

Other parameters can be configured on the selection manager to specify the line width, the color of the selection rectangle used in regular selection mode, or to define whether the selection rectangles are filled or hollow.

```
<jvgf:selectionManager lineWidth="1" lineColor="orange" fillOn="true"/>
```

The selection must be configured separately for the Gantt sheet and Gantt table by using the corresponding `sheetSelectionManager` and `tableSelectionManager` facets.

Listening for changes to the selection

To register a listener that will be called when the selection changes use the following code:

```
<jvgf:selectionManager [...]
  onSelectionChanged="displayProperties(selection)" />
```

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

The JavaScript function can be as follows:

```
// Display the identifier and object type of all the selected objects.
function displayProperties(selection) {
  for (var i = 0; i < selection.length; i++) {
    alert(selection[i].getID()+" "+selection[i].getObjectType());
  }
}
```

The possible values for the `objectType` property are `activity`, `constraint`, `reservation`, and `resource`.

The value of the property ID for activities and resources is the same as the property ID of `IlvActivity` and `IlvResource` in the `IlvGanttModel`. Constraints and reservations have no

such property in the `IlvGanttModel`; the value of their property ID is constructed from the identifiers of the activities and resources they relate to.

Note: You may set a listener and a property accessor on both the `tableSelectionManager` and the `sheetSelectionManager`, but this is not required nor always useful. A selection manager handles only objects available in the corresponding component.

Therefore, in a `ganttView` the selection manager for the Gantt table handles only activities; the selection manager for the Gantt sheet handles both activities and constraints. In this case, listening to changes in the selection only through the `sheetSelectionManager` fits most purposes.

In a `scheduleView` the selection manager for the Gantt table handles only resources; the selection manager for the Gantt sheet handles only reservations. Depending on your needs you may want to listen to changes in the selection for both the `sheetSelectionManager` and `tableSelectionManager`.

Retrieving and setting properties of selected objects

Besides the ID and `objectType` properties of the selected object, you might want to retrieve the properties of the selected objects in the JViews Gantt data model. This can be done by configuring a property accessor on the selection manager used to handle the changes in the selection:

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

with:

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

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

Once the property accessor is defined, you can access the properties of the selected objects in a JavaScript selection listener.

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

```
        alert(selection[i].getObjectProperty(propertiesNames[j]));
    }
}
```

Note: In image mode the mechanism to retrieve properties is not active by default. To activate it, you must force an additional request to the server by setting the `selectionManager` attribute `forceUpdateProperties` to `true`. In regular mode the properties are available without any overhead.

You can also change property values on the client and commit these changes to the server-side model. To do this, first set the `ganttView` as editable:

```
<jvfgf:ganttView editable="true" [...] />
```

Then, to change property values and commit the changes to the server:

```
// Modify a property on the first selected object in the Gantt sheet.
thegantt.getSheetSelectionManager().getSelection()[0].setObjectProperty(
    "propertyName",
    "propertyValue");
// [Other modifications]
// Commit the changes to the server.
thegantt.getSheetSelectionManager().commitSelectionProperties(true, oncompleted,
onfailed);
```

In the example:

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

A property value must be either a JavaScript String or null. For more information on property values see *Predefined properties of Gantt data model objects* and *User-defined properties of Gantt data model objects*. For information on marshalling or unmarshalling the property values, see *Marshalling and unmarshalling property values*.

Marshalling and unmarshalling property values

The property values are exchanged between the server and the client as String values and may also be `null`.

The server uses `IlvConvert` to convert model object property values to String and from a string back to object properties. The following table lists the types specific to JViews Gantt for which converters are installed by default.

Converters registered for marshalling or unmarshalling properties

Type	Converter
java.util.Date	Uses an <code>ilog.views.util.beans.editor.IlvDatePropertyEditor</code>
ilog.views.gantt.IlvConstraintType	Uses an <code>ilog.views.gantt.beans.editor.IlvConstraintTypeEditor</code>
ilog.views.gantt.IlvDuration	Uses an <code>ilog.views.gantt.beans.editor.IlvDurationEditor</code>

Refer to `ilog.views.util.convert.IlvConvert` for information on how to register a converter for a specific Java™ type. Note that you can register either a `java.beans.PropertyEditor` to handle the conversion to and from String or two instances of `IlvConverter`, one to convert from the value type to String and one to convert from String to the value type.

On the client side, no converter is provided. The client is responsible for making sure that the property values sent to the server are valid String values or `null`, such that they can be unmarshalled by the server-side conversion mechanism. The server ignores property values that it cannot unmarshall and notifies the client of the error using the `oncompleted` handler that is defined when `commitSelectionProperties` is called.

Predefined properties of Gantt data model objects

The following table lists the predefined objects of the Gantt data model, their properties, their server-side Java type, and the corresponding client-side values.

Predefined Gantt data model objects and related properties

Object Type	Property Name	Server-side Type	Client-side Value
activity	endTime	java.util.Date	String representing the number of milliseconds elapsed since 1970-01-01 00:00:00 GMT
	id	String	Same as the server-side value.
	name	String	Same as the server-side value.
resource	startTime	java.util.Date	See <code>endTime</code>
	id	String	Same as the server-side value.
	name	String	Same as the server-side value.
constraint	quantity	Float	String value
	constraintType	ilog.views.gantt.IlvConstraintType	Possible values: "End-End", "End-Start", "Start-End",

Object Type	Property Name	Server-side Type	Client-side Value
			"Start-Start".
	fromActivity	ilog.views.gantt. IlvActivity	ID of the IlvActivity
	toActivity	ilog.views.gantt. IlvActivity	ID of the IlvActivity
reservation	activity	ilog.views.gantt. IlvActivity	ID of the IlvActivity
	resource	ilog.views.gantt. IlvResource	ID of the IlvResource

On the server, values of type `java.util.Date` are converted to `String` and back using the `ilog.views.util.beans.editor.IlvDatePropertyEditor`. You can create the corresponding `JavaScript Date` object on the client using the following `JavaScript` code:

```
var date = (value == null) ? null : new Date(parseInt(value));
```

Conversely, to create the property value suitable for committing to the server, use the following `JavaScript` code:

```
var value = (date == null) ? null : date.valueOf().toString();
```

User-defined properties of Gantt data model objects

Gantt data model objects that implement `IlvUserPropertyHolder` may contain user-defined properties. To handle a user-defined property properly on the server, you must:

1. Make sure that the property type has appropriate converters registered in `IlvConvert`. See *Marshalling and unmarshalling property values* for more information.
2. Register the type of the property value in your `ilog.views.gantt.faces.dhtml.component.IlvFacesGanttPropertyAccessor`, as follows:

```
public class ServerBean {
    private IlvFacesGanttPropertyAccessor accessor;
    public ServerBean() {
        accessor = new IlvFacesGanttPropertyAccessor();

        // The user defined property 'myProperty' on activities,
        // is of type MyPropertyType.
        accessor.registerType(IlvActivity.class,
            "myProperty",
            MyPropertyType.class);
    }
}
```

```

    }
    public IlvFacesGanttPropertyAccessor getPropertyAccessor() {
        return accessor;
    }
}

```

By default, all user-defined properties are exported to the client and are settable. You can customize this behavior by extending `IlvFacesGanttPropertyAccessor` and redefining one or all of the methods `getPropertyNames()`, `acceptProperty()`, `acceptGetProperty()`, and `acceptSetProperty()`.

Selecting an object by its identifier

Objects can be selected by their identifiers with the JavaScript method `selectById`.

The object identifiers are controlled by the `IlvGanttSelectionSupport.IdentifierFactory` set on `IlvGanttSelectionSupport`.

The identifiers are built by the default factory implementation:

- ◆ `IlvActivity`, identifier of the activity as returned by `getID()`.
- ◆ `IlvConstraint`, identifier of the `from` and `to` activities, which are separated by a slash; for example, a constraint from the activity A-1 to the activity A-2 gets the identifier A-1/A-2.
- ◆ `IlvResource`, identifier of the resource as returned by `getID()`.
- ◆ `IlvReservation`, identifier of the activity and resource, which are separated by a slash; for example, a reservation of the resource R-1 for the activity A-1 gets the identifier A-1/R-1.

The selectable type of object depends on the type of hierarchy chart displayed:

- ◆ Gantt chart: Only activities and constraints can be selected.
- ◆ Schedule chart: Only resources and reservations can be selected.

For example, you can select one activity in a Gantt chart as follows:

```

thegantt.getTableview().getSelectionManager().selectById("A-1", "activity");

```

This method call deselects the objects currently selected and selects the activity with the identifier A-1. You can extend or reduce the selection by selecting or deselecting an activity as follows:

```

thegantt.getTableview().getSelectionManager().selectById("A-1", "activity",
true);

```

This method call keeps the existing selection and selects the object with the identifier A-1 if it is not already selected. Otherwise, it will deselect it.

Clear the selection

To clear the selection use the JavaScript method `deselectAll`.

For example:

```
thegantt.getTableView().getSelectionManager().deselectAll();
```

Connecting a Gantt chart to a message box

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

```
<jvgf:ganttChartView [...] messageBoxId="messageBox"/>  
<jv:messageBox id="messageBox" [...] />
```

The messages issued are displayed in the message box.

Connecting a chart view to a message box

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

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

The messages issued are now displayed in the message box.

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 Gantt.

```
<jvgf:ganttView [...] >  
  <jvgf:ganttContextualMenu [...] />  
</jvgf:ganttView>
```

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

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

```

    <jv:menuItem label="Select"
               actionListener="#{ganttBean.action}"
               image="images/arrow.gif"
               invocationContext="IMAGE_SERVLET_CONTEXT" />
  </jv:menu>
</jv:gf:ganttContextualMenu>

```

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 Gantt

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

```

<jv:gf:ganttContextualMenu factory="#{bean.factory}" />
<jv:gf:ganttContextualMenu factoryClass="com.xyz.demo.DemoFactory" />

```

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 Gantt

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

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 Gantt

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

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 Gantt 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:

```
<jv:gf:ganttView id="gantt" [...] />
```

will be rendered as the following JavaScript code:

```
gantt = new IlvHierarchyChartViewProxy ('gantt', ' ...');
```

Note: See the documentation of the Java API of each renderer to know which JavaScript proxy will be generated for this component.

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

For example, the following code defines two buttons that install respectively a scroll interactor and a row expand or collapse interactor on the table of a `ganttView`.

```
<jv:imageButton [...] onclick="gantt.setTableInteractor(tableScrollInteractor)
" />
<jv:imageButton [...] onclick="gantt.setTableInteractor(tableExpandInteractor)
" />
<jv:gf:ganttView id="gantt" [...] />
```

At rendering time, an `IlvHierarchyChartViewProxy` JavaScript object is created that is accessible through the `gantt` JavaScript variable. Then, since `rowExpandCollapseInteractor` and `tableScrollInteractor` JavaScript objects have been created in the same way, you can directly set one of these interactors with the `setInteractor` method.

It is possible to set the interactor of the table and the sheet views in one call:

```
<jv:imageButton [...]
  onclick="gantt.setInteractors(tableScrollInteractor, sheetScrollInteractor)
" />
```

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

For further information about available JavaScript objects, see [Javascript API](#), the JavaScript API reference documentation of JViews Gantt.

Contexts for actions on the Gantt Chart 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 the Context in Which 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 hierarchy 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 Gantt

To expand an activity in a Gantt chart view, a select hierarchy node interactor must be installed on the Gantt chart view. The `value` property of the interactor holds the `IlvHierarchyNode` that was clicked. (The hierarchy node is an activity in a Gantt chart or a resource in a Schedule chart.) Thus, a `valueChangeListener` can be registered to handle the selection event.

Installing a select hierarchy node interactor and a listener

```
<jvgf:nodeSelectInteractor id="expand"
    valueChangeListener="#{gantttBean.expandAllRows}"
                        invocationContext="JSF_CONTEXT"/>
<jvgf:ganttView id="gantttView" sheetInteractorId="expand" [...] />
```

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 expandAllRows (ValueChangeEvent event) {
    IlvActivity activity = (IlvActivity) event.getNewValue();
    if (activity != null) {

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

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

        //Retrieve the IlvHierarchyChart wrapped by the JSF component.
        IlvHierarchyChart chart = jsfView.getChart();

        if (chart.isRowExpanded(activity)) {
            chart.collapseRow(activity);
        } else {
            chart.expandAllRows(activity);
        }
    }
}
```

```
}  
}
```

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

```
<jvpcf:chartSelectInteractor id="selectInteractor"  
    valueChangeListener="#{demoBean.pointSelected}"  
  
    pickingMode="item"  
        invocationContext="JSF_CONTEXT">  
<jvpcf: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 Gantt)

```
<jvvgf:nodeSelectInteractor id="expand"
    valueChangeListener="#{ganttbbean.expandAllRows}"
                        invocationContext="IMAGE_SERVLET_CONTEXT"/>
<jvvgf:ganttView id="ganttbview" sheetInteractorId="expand" [...] />
```

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 Gantt:

- ◆ 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 `IlvHierarchyChart` object as shown in *Java code of the value-change event*.

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 underlying 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 Gantt

Using JavaScript actions in a JSP page

```
[...]  
<jv:rowExpandCollapseInteractor id="tableExpand" />  
<jv:rowExpandCollapseInteractor id="sheetExpand" />  
  
<jv:imageButton [...] onclick="#{ganttBean.setExpandAction}" />  
<jv:ganttView id="gantt" [...] />  
[...]
```

Using JavaScript actions in a managed bean

```
public class GanttBean {  
[...]  
    private String setExpandAction;  
    public GanttBean(){  
        setExpandAction =  
            IlvFacesUtil.encodeJavaScriptVariables("#{gantt}.setInteractors({  
                tableExpand, ${sheetExpand})");  
    }  
    public String getSetExpandAction(){  
        return setExpandAction;  
    }  
[...]  
}
```

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 Gantt

```
<jvgf ganttView [...] servlet=  
    "ilog.views.gantt.faces.dhtml.servlet.IlvFacesGanttServlet />"
```

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 Gantt application examples configured for use with Facelets or Trinidad, see `<install-dir>/jviews-gantt8.6/codefragments/jsf-gantt-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`.

For JViews Gantt, the contextual menu element is `ganttContextualMenu`.

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.

Overview

Gives an overview of the thin-client approach in JViews Gantt.

Gantt Thin-Client Web Architecture

Explains how the Gantt thin-client Web application support uses the *Java servlet* technology to deliver information from a Gantt chart server-side application.

Getting Started With the Gantt Thin Client: An Example

Provides an example to help you get started with thin client development.

Developing the server side

Explains how to develop a server side JViews Gantt thin-client application.

Developing the client side

After creating the server side of your Gantt Web application, you create the client side. The Gantt thin-client support allows you to easily build a client based on Dynamic HTML that will run on Web browsers that support DHTML. You build an HTML Web page for the DHTML client using predefined JavaScript™ components.

Adding client/server interactions

The JViews Gantt 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 name of an activity that appears on the generated image. Part of this action, clicking the image to select the activity, must be done on the client side. Changing the name of the activity in the Gantt data model 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.

The `IlvGanttServlet` and `IlvGanttServletSupport` classes

Describes how to create a servlet and how the servlet responds to different requests.

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 Gantt 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.

Overview

Developing with the JViews Gantt SDK discusses how you can use JViews Gantt on the client side where you develop Java™ applets or applications. You can also use JViews Gantt on the server side. Some Web applications require that the client stay very light, with most of the functionality residing in the server. JViews Gantt thin-client support allows you to create such types of applications easily. You can use the power of IBM® ILOG® JViews Gantt to build Gantt or Schedule charts on the Web server. You can then use JViews Gantt thin-client support on your Web browser to display and interact with those images created by the server.

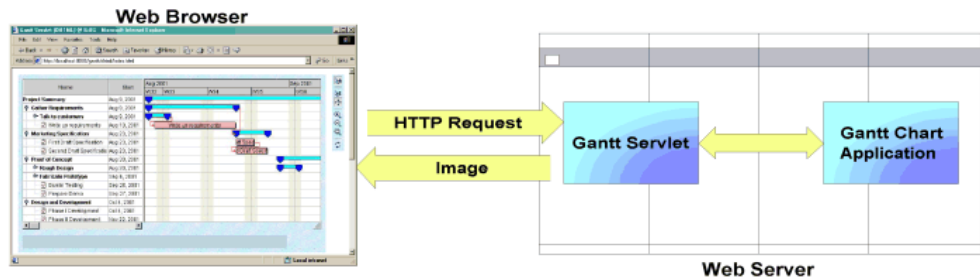
This section explains how to use JViews Gantt packages and classes on both the server side and the client side.

- ◆ *Gantt Thin-Client Web Architecture*
- ◆ *Getting Started With the Gantt Thin Client: An Example*
- ◆ *Developing the server side*
- ◆ *Developing the client side*
- ◆ *Adding client/server interactions*
- ◆ *The IlvGanttServlet and IlvGanttServletSupport classes*

Gantt Thin-Client Web Architecture

The Gantt thin-client Web application 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>. This site also provides information about the Web servers that support Java servlets.

For the predefined Gantt thin client, the content created by the servlet is primarily a JPEG or PNG image. The servlet generates the images from a Gantt chart server-side application that is almost identical to the client-side Gantt chart applications discussed in previous sections. The servlet acts as an intermediate layer. It interprets the HTTP requests from the thin client running in the user's browser, generates images of the Gantt chart server-side application, and delivers the images in HTTP responses back to the client. In turn, the Gantt chart server-side application may obtain the scheduling information that it displays from XML files, databases, or other application-specific data. This basic architecture is illustrated in *Gantt thin-client Web application architecture*:



Gantt thin-client Web application architecture

The JViews Gantt thin-client support contains the following:

- ◆ An abstract servlet class that can generate images from a Gantt chart display.
- ◆ A set of browser-independent Dynamic HTML scripts written in JavaScript™ that can be used on the client side to display and interact with the images created on the server side.

Getting Started With the Gantt Thin Client: An Example

Provides an example to help you get started with thin client development.

In this section

Creating a Gantt thin-client application

Describes the steps necessary to create a Gantt thin-client application

The Gantt Servlet Example

Gives an overview of the Gantt Servlet example.

Installing and Running the Gantt Servlet Example

Describes the requirements to install and run this sample.

Creating a Gantt thin-client application

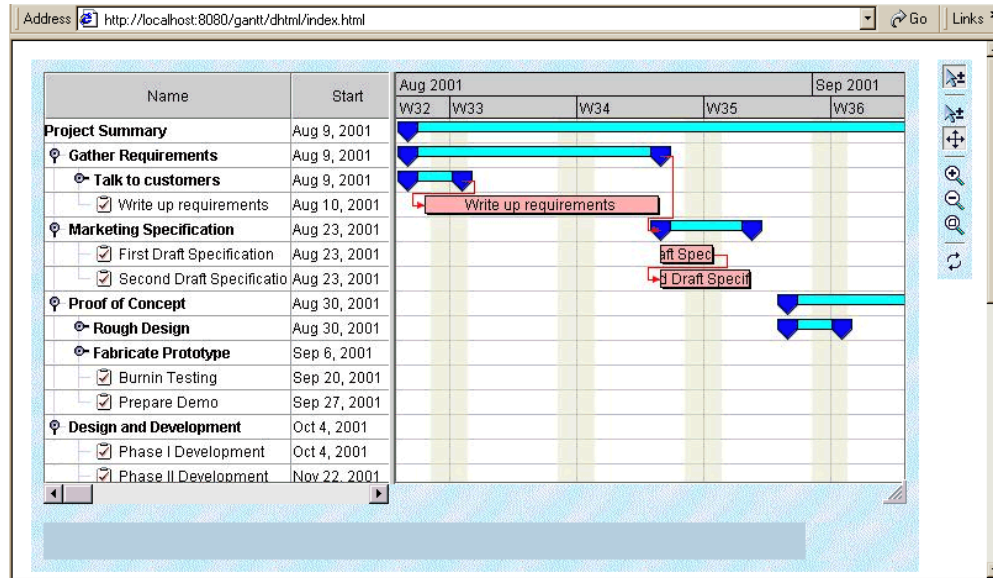
Creating a Gantt thin-client application consists of two steps: developing the server side and developing the client side. The Gantt Servlet example, provided with the distribution, illustrates these steps in this section.

The Gantt Servlet Example

The Gantt Servlet example can be found in the following directory:

```
<installdir>/jviews-gantt86/samples/servlet
```

This example allows you to show a standard Gantt chart of scheduling information in a thin-client context.



The Gantt Servlet Example

The Gantt Servlet example is composed of the following:

The Server Side

The server side consists of three Java™ files located in the directory:

```
<installdir>/jviews-gantt86/samples/servlet/src
```

These files are:

File	Description
<code>GanttChartServlet.java</code>	A servlet that produces JPEG images from a standard <code>IlvGanttChart</code> component.
<code>SimpleProjectDataModel.java</code>	A Gantt data model that contains the project scheduling information.
<code>BasicServletSupport.java</code>	A basic implementation of <code>IlvGanttServletSupport</code> that handles all the HTTP requests for the servlet.

The DHTML client

The DHTML client consists of:

- ◆ The HTML starting page:

`<installdir>/jviews-gantt86/samples/servlet/web/index.html`

- ◆ The set of JViews Framework common JavaScript™ DHTML components, located in:

`<installdir>/jviews-framework86/lib/thinclient/javascript/`

and the images needed for these components in:

`<installdir>/jviews-framework86/lib/thinclient/javascript/images`

- ◆ The set of Gantt JavaScript DHTML components, located in:

`<installdir>/jviews-gantt86/lib/thinclient/javascript/gantt`

Installing and Running the Gantt Servlet Example

Running the Gantt Servlet example requires a Web server and a Web browser that support Dynamic HTML. The Web server must support the Servlet API 2.1 or later.

This sample is compatible with the browsers and browser versions listed in the Release notes.

The Gantt Servlet example contains a WAR file (Web Archive):

that allows you to easily install the example on the Web server of your choice. You can check the latest list of servers that support servlets at:

<http://java.sun.com/products/servlet/industry.html>

For your convenience, the Apache Tomcat™ Web server is supplied with the JViews Gantt distribution. The Gantt Servlet example is pre-installed in Tomcat and is ready to run. TOMCAT is the official reference implementation of the Servlet and JSP™ specifications. To get more information on Tomcat go to *<http://jakarta.apache.org/tomcat/>*.

The steps for running the Tomcat Web server supplied with the JViews Gantt installation can be found at:

◆ `<installdir>/jviews-gantt86/samples/servlet/gantt-thinclient.war`

◆ Starting the samples

Note: If you are running on Microsoft® Windows® then you will find menu items in the Windows "start" menu to start and stop the Tomcat server.

1. Launch a Web browser and open the page: `http://localhost:8080/gantt-thinclient`
2. For additional details, consult the instructions on how to run the IBM® ILOG® JViews Gantt samples in:

Starting the samples

Your browser then shows the Gantt Servlet example.

Developing the server side

Explains how to develop a server side JViews Gantt thin-client application.

In this section

Key classes and their associations

Describes key classes used in a server side JViews Gantt thin-client application.

The servlet support class

Describes the main class used in the Gantt Servlet example.

Multithreading issues on the server side

Explains how to use non-multithreaded Swing GUI components in a multithreaded Web server run-time environment.

The servlet class

Explains the function of the servlet class.

Answering HTTP requests

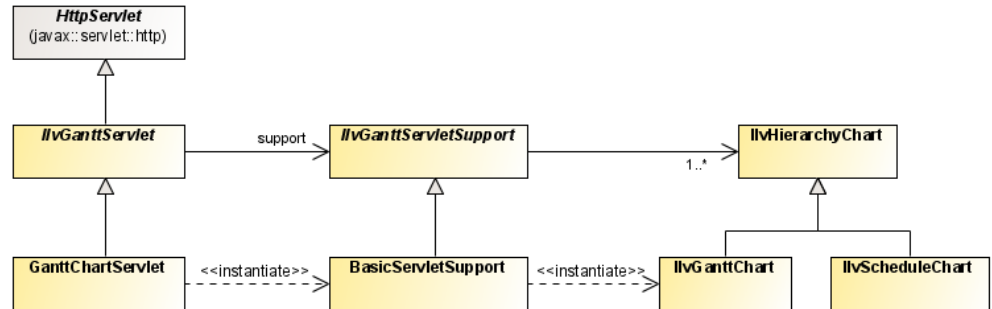
Describes what is returned when you call the server side of the Gantt chart Web application.

Key classes and their associations

The server side of a JViews Gantt thin-client application is composed of two main parts:

- ◆ The JViews Gantt application itself, which can be any type of Gantt chart or Schedule chart built upon the JViews Gantt components and APIs, and
- ◆ A servlet that interprets requests from the client to generate images of the chart.

The following figure shows an overview of the key classes and their associations.



Overview of key server-side classes

The server-side classes are colored to indicate their packaging:

- ◆ The yellow class, `HttpServlet`, is part of the standard Java™ Servlet API. It is located in the `javax.servlet.http` package and is the abstract base class for all HTTP servlet implementations.
- ◆ The blue classes are members of the JViews Gantt API. The abstract classes `IlvGanttServlet` and `IlvGanttServletSupport` belong to the `ilog.views.gantt.servlet` package. `IlvGanttServlet` is an abstract servlet that responds to HTTP requests to generate images of a Gantt chart or a Schedule chart. `IlvGanttServlet` is a very simple class that delegates all its real work to an instance of `IlvGanttServletSupport`. This allows you to easily integrate the full capabilities of the Gantt server-side classes into your own servlet implementations.
- ◆ The green classes belong to the Gantt Servlet example and are located in the file:

```
<installdir>/jviews-gantt86/samples/servlet/src/GanttChartServlet.java
```

The class `GanttChartServlet` is the concrete servlet implementation for the server side of the example. Its concrete inner support class, `BasicServletSupport`, generates images of a standard Gantt chart in response to HTTP requests.

The subsequent sections explain how the server side is built in the Gantt Servlet example:

- ◆ *The servlet support class*
- ◆ *Multithreading issues on the server side*
- ◆ *The servlet class*

◆ *Answering HTTP requests*

The servlet support class

The Gantt Servlet example displays a standard Gantt chart containing project scheduling information. The `IlvGanttServletSupport` class does all the work on the server side to generate images of the chart in response to HTTP requests. The concrete implementation for this example is the `IlvBasicServletSupport` inner class, located in the file:

```
<installdir>/jviews-gantt86/samples/servlet/src/GanttChartServlet.java
```

The getChart method

The method:

```
createServletSupport public IlvHierarchyChart getChart(HttpServletRequest,
IlvServletRequestParameters) throws ServletException
```

is the only abstract method of the `IlvGanttServletSupport` class. It should return the `IlvGanttChart` or `IlvScheduleChart` instance that will be used to satisfy an HTTP request. The request is given as a parameter to the `getChart` method, so it is possible to provide charts to the client that are session-specific. The servlet support class of the example has been simplified to use a single Gantt chart instance to satisfy all HTTP requests. This means that every client will see the same data.

Details of the code sample

The code of the example starts with `include` statements:

1. First, the `import` statements required to use the Java™ Servlet API:

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

2. Then the `import` statements that are required for JViews Gantt and the JViews Gantt server-side classes:

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

The servlet support class of the example is very simple and consists of only two methods:

```
public class GanttServletSupport extends IlvGanttServletSupport
{
    private IlvHierarchyChart _chart;

    /**
     * Creates the Gantt chart that will be used by the servlet to satisfy HTTP
     * requests.
     */
    private IlvHierarchyChart createChart(IlvGanttModel ganttModel)
    {
        IlvHierarchyChart chart = new IlvGanttChart();
        chart.setGanttModel(ganttModel);
    }
}
```

```

        ... chart customizations ...
        return chart;
    }

    /**
     * Returns the chart used for the specified request. This implementation
     * always returns the same chart.
     * @param request The current HTTP request.
     * @param params The parameters parsed from the request.
     */
    public IlvHierarchyChart getChart(HttpServletRequest request,
                                      IlvServletRequestParameters params)
        throws ServletException
    {
        synchronized(this) {
            if (_chart == null) {
                _chart = createChart(new SimpleProjectDataModel());
            }
        }
        return _chart;
    }
}

```

As you can see, the steps necessary to create a chart on the server side are almost identical to those discussed in earlier sections for developing client-side Java applications and applets. In summary, you need to:

1. Create a concrete subclass of `IlvGanttServletSupport`.
2. Implement the `getChart` method to return an instance of `IlvGanttChart` or `IlvScheduleChart`.
3. Connect the chart to your application data model and customize the appearance of the chart as you desire.

Multithreading issues on the server side

Using GUI components, such as the `IlvGanttChart` and `IlvScheduleChart` involves threading issues on the server side. The Web server run-time environment is inherently multithreaded. However, the Gantt chart components, like all Swing GUI components, are not multithread-safe. There is also a further design constraint of Swing GUI components. Namely, after the Web server has sent an image of a chart to the client for the first time, all modifications to the visual properties of the chart must be performed on the AWT event dispatch thread. The AWT event dispatch thread will never be the same thread that the HTTP request is being serviced on.

In general, the `IlvGanttServletSupport` base class handles all these threading issues for you. It ensures that all requests to modify a chart and generate its image are moved from the HTTP request thread onto the AWT event dispatch thread as necessary. In the `createChart` method of the servlet support class, you are able to customize the visual properties of the chart on the HTTP request thread because the chart has not been sent to the client yet. However, note the use of the `synchronized` block in the `getChart(javax.servlet.http.HttpServletRequest, ilog.views.gantt.servlet.IlvServletRequestParameters)` method. This is necessary to ensure that only a single chart instance is ever created in the multithreaded Web server environment.

The servlet class

The `IlvGanttServlet` class is a simple HTTP servlet implementation that delegates all its work to its associated support class. It contains a single abstract method:

```
createServletSupport()
```

This method must return the single support instance that will service the HTTP requests sent to the servlet. The implementation of this class for the example is located in the file: **<installdir>/jviews-gantt86/samples/servlet/src/GanttChartServlet.java**

That implementation consists of only the `createServletSupport` method:

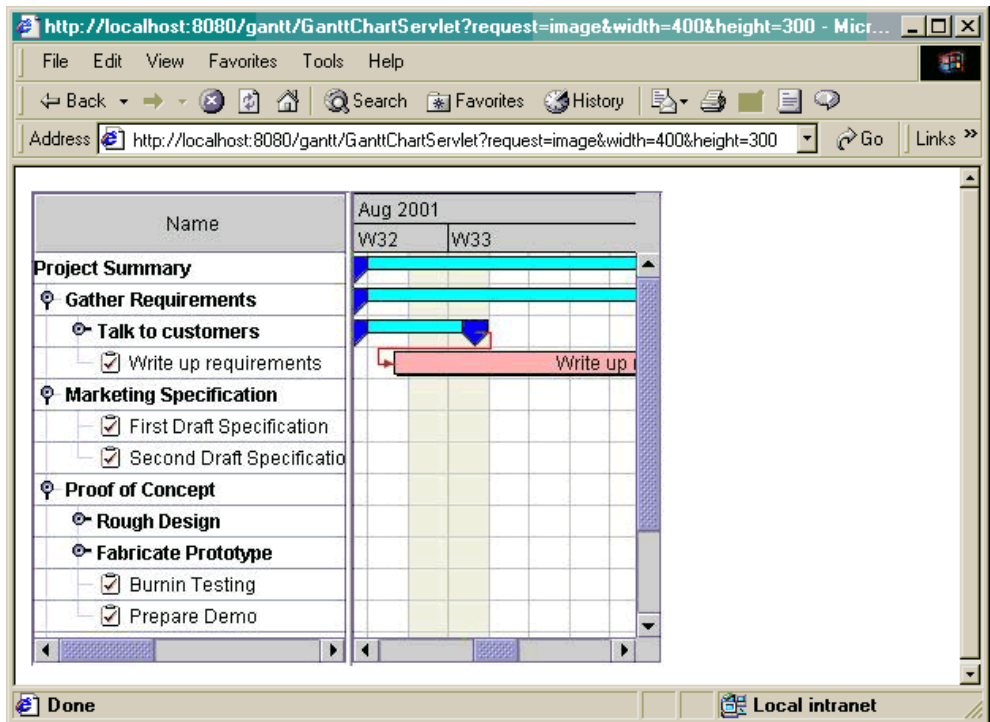
```
public class GanttChartServlet extends IlvGanttServlet
{
    /**
     * Creates the servlet support object to which this servlet delegates HTTP
     * request handling.
     */
    protected IlvGanttServletSupport createServletSupport()
    {
        IlvGanttServletSupport support = new BasicServletSupport();
        ... customize the support class ...
        return support;
    }
}
```

Answering HTTP requests

Creating the server side of the Gantt chart Web application is very simple. The servlet can now answer HTTP requests from a client by sending JPEG images of the chart. If the Apache Tomcat™ server is running, you can try typing the following HTTP request in your Web browser:

```
http://localhost:8080/gantt-thinclient/  
GanttChartServlet?request=image&width=400&height=300
```

This produces the following image:



This request asks the servlet named `GanttChartServlet` to produce an image of size 400 x 300 showing the entire `IlvGanttChart` component. In most cases, you do not have to know the servlet parameters because the client-side Dynamic HTML objects provided by JViews Gantt takes care of the HTTP requests for you.

Developing the client side

After creating the server side of your Gantt Web application, you create the client side. The Gantt thin-client support allows you to easily build a client based on Dynamic HTML that will run on Web browsers that support DHTML. You build an HTML Web page for the DHTML client using predefined JavaScript™ components.

In this section

Developing a Dynamic HTML client

Explains the advantages of the Dynamic HTML client and the components supplied to develop them.

The DHTML client for the Gantt Servlet example

Explains how to create a Dynamic HTML client.

The Popup menu in JavaScript

Describes the JavaScript component for the popup menu.

Developing a Dynamic HTML client

The static nature of HTML limits the interactivity of Web pages. Dynamic HTML allows you to create Web pages that are more interactive and engaging. It gives content providers new controls and allows them to manipulate the contents of HTML pages through scripting. To learn more about Dynamic HTML, you can search for the following items on Web sites:

- ◆ The Microsoft® Web Workshop within:

`http://msdn.microsoft.com`

IBM® ILOG® JViews Gantt provides a set of browser-independent Dynamic HTML components written in JavaScript™ that allow you to build your DHTML pages very easily.

- ◆ JViews Framework common JavaScript DHTML components are located in:

`<installdir>/jviews-framework86/lib/thinclient/javascript`

- ◆ JViews Gantt JavaScript DHTML components

`<installdir>/jviews-gantt86/lib/thinclient/javascript/gantt`

Warning: Keep in mind that not all versions of Web browsers support DHTML. See the Release notes for the browsers and browser versions that the IBM® ILOG® JViews DHTML scripts have been tested on.

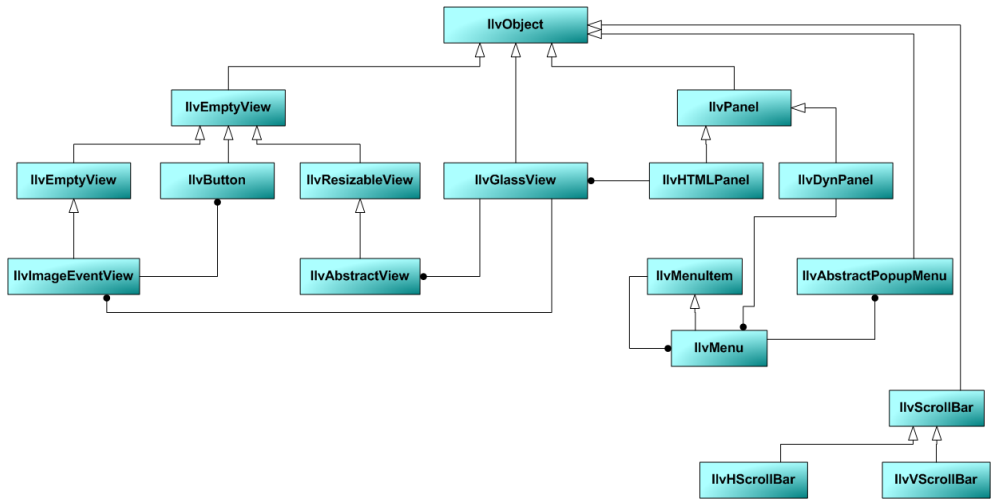
Common DHTML components

See also *The IlvView JavaScript Component in Advanced Features of IBM ILOG JViews Framework*.

The common JavaScript™ DHTML components are located in the directory:

`<installdir>/jviews-framework86/lib/thinclient/javascript`

Here is an overview of the common DHTML component classes and their relationships:



Common DHTML Components

Here is the list of common JavaScript files and a brief description of each:

Common DHTML Script Files

Script File	Description
<code>IlvAbstractPopupMenu.js</code>	Defines the <code>IlvAbstractPopupMenu</code> , <code>IlvMenu</code> , and <code>IlvMenuItem</code> classes that are the base classes of popup menus.
<code>IlvAbstractView.js</code>	Defines the <code>IlvAbstractView</code> class, the base class for <code>IlvGanttComponentView</code> .
<code>IlvButton.js</code>	Defines the <code>IlvButton</code> class, a simple DHTML button.
<code>IlvEmptyView.js</code>	Defines the class <code>IlvEmptyView</code> , the base class for all view components that have a size and position on the HTML page.
<code>IlvEvaluatorView.js</code>	Defines a very simple JavaScript debugging window that can be added to your Web page.
<code>IlvGlassView.js</code>	Defines the <code>IlvGlassView</code> class.
<code>IlvImageView.js</code>	Defines the <code>IlvImageView</code> and <code>IlvImageEventView</code> classes.
<code>IlvInteractor.js</code>	Defines the <code>IlvInteractor</code> class, the base class for all view interactors.
<code>IlvInteractorButton.js</code>	Defines the <code>IlvInteractorButton</code> class, a subclass of <code>IlvButton</code> that can set an interactor on a view.
<code>IlvResizableView.js</code>	Defines the <code>IlvResizableView</code> class, the base class for all view components that can be interactively resized. This is the base class for <code>IlvGanttView</code> .
<code>IlvScrollbar.js</code>	Defines the DHTML scroll bar classes <code>IlvScrollbar</code> , <code>IlvVScrollbar</code> , and <code>IlvHScrollbar</code> .
<code>IlvToolBar.js</code>	Defines the <code>IlvToolBar</code> class, a DHTML toolbar that can contain <code>IlvButtons</code> .
<code>IlvUtil.js</code>	Dynamic HTML tools and functions used by other scripts. This file must always be included. This file defines the <code>IlvObject</code> and <code>IlvPanel</code> classes.

The full reference documentation of each component can be found in the Dynamic HTML Component Reference located in:

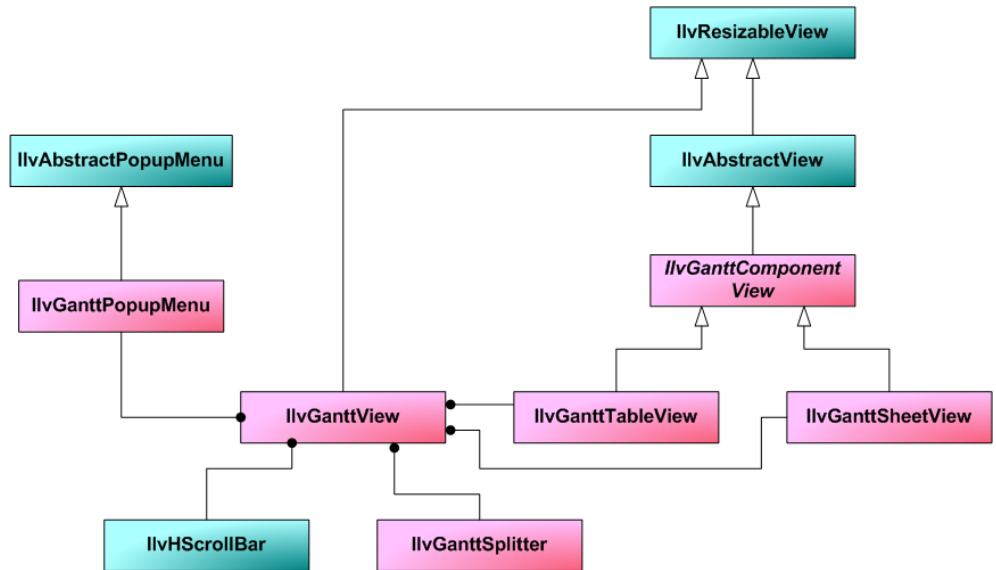
`<installdir>/jviews-framework86/doc/html/refjs_fwork/index.html`

Gantt DHTML components

The JViews Gantt JavaScript DHTML components are located in the directory:

`<installdir>/jviews-gantt86/lib/thinclient/javascript/gantt`

Here is an overview of the Gantt DHTML component classes and their relationships:



Gantt DHTML Components

Here is the list of JViews Gantt JavaScript files and a brief description of each:

Gantt DHTML Script Files

Script File	Description
IlvGanttPopupMenu.js	Defines the <code>IlvGanttPopupMenu</code> class that is the Gantt-specific implementation of <code>IlvAbstractPopupMenu</code> .
IlvGanttView.js	Defines the main Gantt view classes <code>IlvGanttView</code> , <code>IlvGanttComponentView</code> , <code>IlvGanttTableView</code> , and <code>IlvGanttSheetView</code> .
IlvGanttTableScrollInteractor.js	Defines the class <code>IlvGanttTableScrollInteractor</code> , an interactor that lets you pan and scroll an <code>IlvGanttTableView</code> .
IlvGanttSheetScrollInteractor.js	Defines the class <code>IlvGanttSheetScrollInteractor</code> , an interactor that lets you pan and scroll an <code>IlvGanttSheetView</code> .
IlvRowExpandCollapseInteractor.js	Defines the class <code>IlvRowExpandCollapseInteractor</code> , an interactor that lets you expand and collapse rows in an <code>IlvGanttTableView</code> or an <code>IlvGanttSheetView</code> .
IlvRowSelectInteractor.js	Defines the class <code>IlvRowSelectInteractor</code> , an interactor that allows you to select rows in an <code>IlvGanttTableView</code> or an <code>IlvGanttSheetView</code> .

The full reference documentation of each component can be found in the Dynamic HTML Component Reference located in:

`<installdir>/jviews-gantt86/doc/html/refjsgantt/index.html`

The DHTML client for the Gantt Servlet example

You will now create a Dynamic HTML client for the Gantt Servlet example, starting with a very simple example and including most of the DHTML components. The full HTML file for the Gantt Servlet example is located in:

```
<installdir>/jviews-gantt86/samples/servlet/webpages/index.html
```

This section covers:

- ◆ *Directory structure of the Web application*
- ◆ *The IlvGanttView DHTML component*
- ◆ *The message panel*
- ◆ *Interactively resizing the Gantt view*
- ◆ *Decorative panels*
- ◆ *IlvToolBar and IlvButton*
- ◆ *IlvGanttSheetScrollInteractor*
- ◆ *IlvRowExpandCollapseInteractor*
- ◆ *IlvInteractorButton*

Directory structure of the Web application

Before you start using the Gantt DHTML components to build the client, you must first decide on the directory structure that the users will see when they visit your Web application with their browser. This structure does not, and should not, match the location of the example and JavaScript files in the IBM® ILOG® JViews Gantt distribution. The Gantt Servlet example is deployed to use the following directory structure:



Directory Structure of the Web Application

The Ant build file for the Gantt Servlet example:

```
<installdir>/jviews-gantt86/samples/servlet/build.xml
```

creates this directory structure in the `gantt-thinclient.war` Web Archive.

The IlvGanttView DHTML component

The `IlvGanttView` component (located in the `IlvGanttView.js` file) is the main Gantt DHTML component. This component queries the servlet and displays the resulting image of the chart. The steps are as follows:

◆ *Importing the JavaScript files*

◆ *Creating the Gantt View*

◆ *Defining JavaScript functions*

Importing the JavaScript files

First, you must include the JavaScript files that are required to use the `IlvGanttView` component:

- ◆ `IlvUtil.js`
- ◆ `IlvEmptyView.js`
- ◆ `IlvImageView.js`
- ◆ `IlvGlassView.js`
- ◆ `IlvResizableView.js`
- ◆ `IlvAbstractView.js`
- ◆ `IlvScrollbar.js`
- ◆ `IlvGanttView.js`

Here is a simple HTML page that creates an `IlvGanttView` object:

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

<script TYPE="text/javascript">
function init()
{
    chartView.init();
}

function handleResize()
{
    if (document.layers)
        window.location.reload()
}
</script>
```

```

</script>
<body onload="init()" onunload="ilvDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet.
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.toHTML();
</script>
</body>
</html>

```

The example starts with importing the necessary 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/IlvScrollbar.js"></script>
<script TYPE="text/javascript" src="script/IlvGanttView.js"></script>

```

The JavaScript files must be placed in the head of the page. Note that the scripts are included from the relative `script` subdirectory. Remember that when you build the Web application, the HTML Web pages will be placed in the upper directory and the scripts will be in the `script` directory (see *Gantt DHTML Components*).

Creating the Gantt View

In the body of the page, you create an `IlvGanttView` located in (25, 25) on the HTML page. The size is 350 x 700. This view displays images produced by the servlet `GanttChartServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

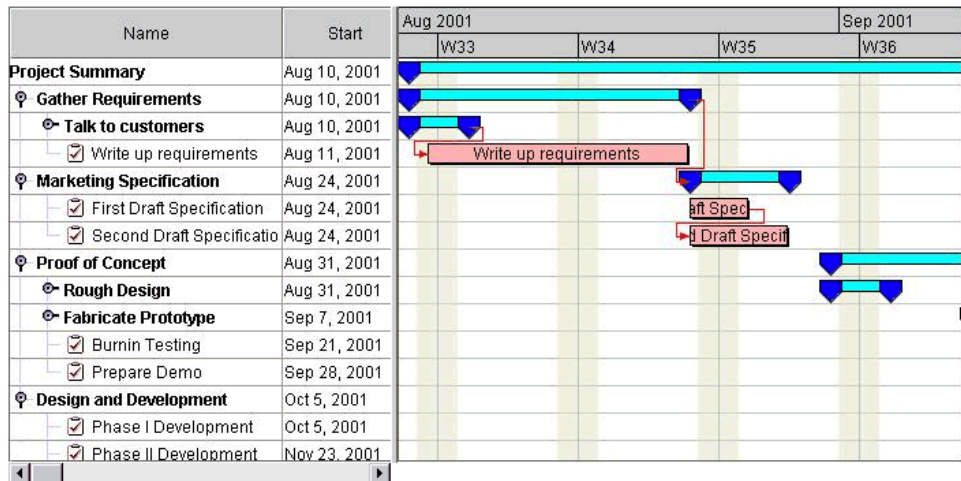
Defining JavaScript functions

This example also defines two JavaScript functions:

- ◆ The `init` function, called on the `onload` event of the page, initializes the `IlvGanttView` 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 `ilvDispose` function must be called in the `onunload` event of the HTML page. This function disposes of all the resources acquired by the DHTML components.

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



The message panel

You will now add a Dynamic HTML panel below our main view. A DHTML panel is an area of the page that can contain some HTML content. You will use the DHTML panel to display status messages as the user interacts with the JViews Gantt view. You create the message panel using the class `IlvHTMLPanel`, defined in the `IlvUtil.js` file.

The body of the page is now:

```
<body onload="init()" onunload="ilvDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet.
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.toHTML();
```

```

var messagePanel = new IlvHTMLPanel('');
messagePanel.setBackgroundColor('#B6D5DA');
messagePanel.setVisible(true);
chartView.setMessagePanel(messagePanel);

var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop() + chart.getHeight() + 15,
                           chart.getWidth(),
                           45);
}
layoutPage(chartView);
chartView.addSizeListener(layoutPage);
</script>
</body>

```

Note that the class `IlvHTMLPanel` does not have a `toHTML` method, it generates its HTML content immediately from within its constructor. Also, the `IlvHTMLPanel` is initially hidden. You must explicitly call its `setVisible` method to show it on the page. These are the main differences between the DHTML “view” components and the DHTML “panel” components.

In this example, we intend to make the main Gantt view interactively resizable. To this effect, there is a `layoutPage` function that positions the message panel relative to the current size and position of the main Gantt view:

```

var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop() + chart.getHeight() + 15,
                           chart.getWidth(),
                           45);
}

```

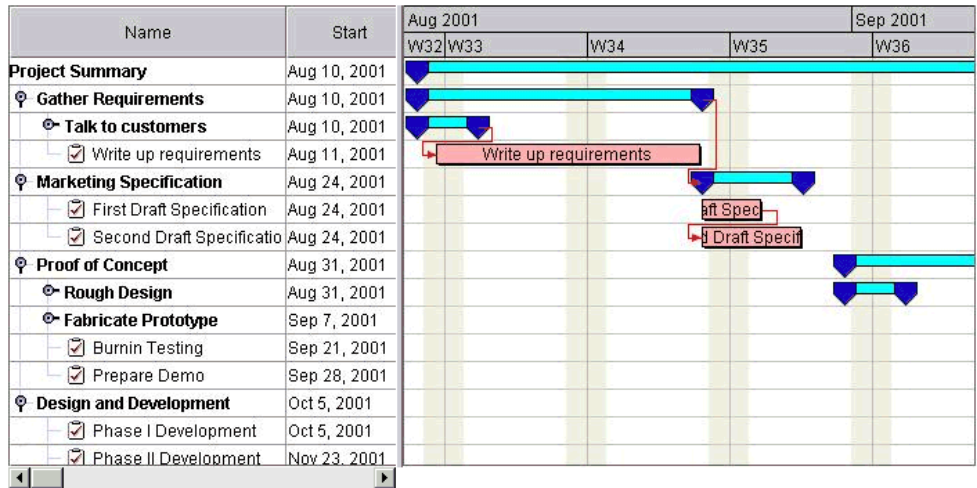
The `layoutPage` function is then called to perform the initial arrangement of the components:

```
layoutPage(chartView);
```

The method `addSizeListener` is called to listen for resize events on `layoutPage` from the `IlvGanttView`:

```
chartView.addSizeListener(layoutPage);
```


The Web page now looks like this:



Interactively resizing the Gantt view

You will now make the main Gantt view interactively resizable by calling the `setResizable` method:

```
var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
chartView.setServletURL(servletName);
chartView.setResizable(true);
chartView.toHTML();
```

Our `IlvGanttView` now displays a resize tool at its lower right corner: 

You can now click and drag on the tool to interactively resize the Gantt view. When the resize operation completes, the `layoutPage` method is invoked, and the position of the message panel is updated to match that of the main view.

Decorative panels

Next, you will add some decorative panels around our main Gantt view to improve the appearance of the Web page. You `IlvHTMLPanel` use to display a tiled image pattern as a background frame:

```
var backgroundPanel = new IlvHTMLPanel('');
backgroundPanel.setBackgroundImage(ilvImagePath + 'skybg.jpg');
```

```
backgroundPanel.setBackgroundColor('#909090');
backgroundPanel.setVisible(true);
```

The background panel must be created before the `IlvGanttView.toHTML` method is invoked. DHTML components have an implied z-order in the browser that is determined by the order in which their HTML code is created in the page body. The `IlvHTMLPanel` component creates its HTML code in its constructor and the `IlvGanttView` component creates its HTML code in its `toHTML` method. By placing the background panel before the Gantt view in the page body, you ensure that the panel will appear behind the Gantt view.

The `ilvImagePath` variable, used to define the tiled image for the background panel, is a global variable defined in `IlvUtil.js`. It contains the path to the images used by the script files. Its default value is `script/images`, which is the location of the image files relative to the Web pages in the Web application.

You will also add a small company logo to display on the right side of the message panel. You use an `IlvImageView` component instead of an `IlvHTMLPanel` because you do not want to tile the logo image:

```
var logoPanel = new IlvImageView(0, 0, 67, 30,
                                ilvImagePath+'ilog-small.gif');
logoPanel.toHTML();
```

The `IlvImageView` component has the additional advantage of remaining hidden until its image is loaded. This is important for a nice appearance when the images take some time to download from the server due to image size or network latencies. Normally, if an image has not been loaded from the server yet, the browser will display a box with a red "X" in



The `IlvImageView` component avoids this effect and remains invisible until the image is available to display.

Finally, you must update the `layoutPage` method to properly arrange the new panels:

```
var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop()+chart.getHeight()+15,
                           chart.getWidth()-logoPanel.getWidth()-15,
                           logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop() - 10,
                              chart.getWidth()+ 20,
                              messagePanel.getTop() +
                                messagePanel.getHeight() + 20 -
                                chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft() +
                          messagePanel.getWidth() + 15,
                          messagePanel.getTop());
}
```

The body of the HTML file now looks like this:

```

<body onload="init()" onunload="ilvDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var backgroundPanel = new IlvHTMLPanel('');
  backgroundPanel.setBackgroundImage(ilvImagePath + 'skybg.jpg');
  backgroundPanel.setBackgroundColor('#909090');
  backgroundPanel.setVisible(true);

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.setResizable(true);
  chartView.toHTML();

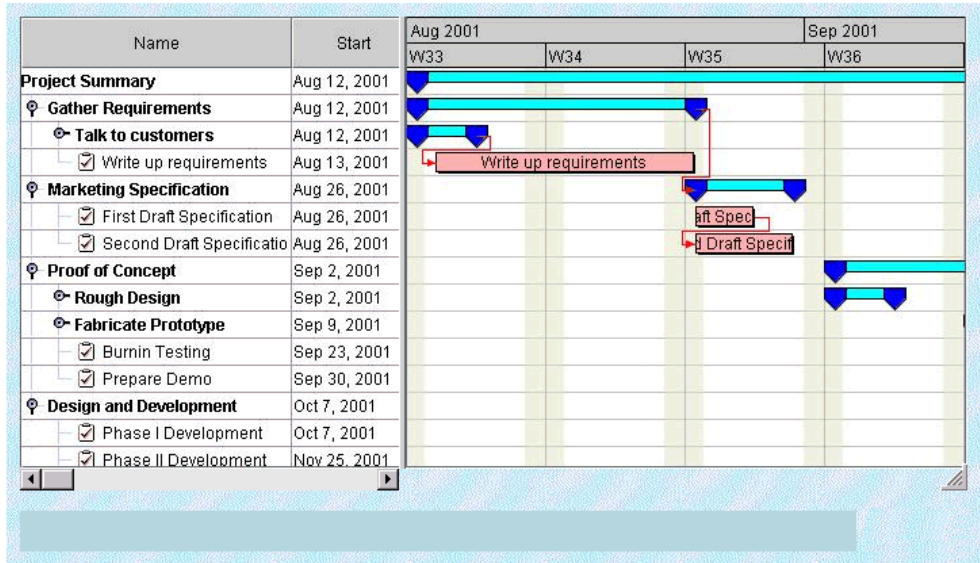
  var messagePanel = new IlvHTMLPanel('');
  messagePanel.setBackgroundColor('#B6D5DA');
  messagePanel.setVisible(true);
  chartView.setMessagePanel(messagePanel);

  var logoPanel = new IlvImageView(0, 0, 67, 30, ilvImagePath+'ilog-small.gif')
;
  logoPanel.toHTML();

  var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                          chart.getTop() + chart.getHeight() + 15,
                          chart.getWidth() - logoPanel.getWidth() - 15,
                          logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop() - 10,
                              chart.getWidth() + 20,
                              messagePanel.getTop() +
                                messagePanel.getHeight() + 20 -
                                chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft() + messagePanel.getWidth() + 15,
                          messagePanel.getTop());
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);
</script>
</body>

```

You should now see the following Web page:



IlvToolBar and IlvButton

The `IlvButton` class is a simple DHTML button component that allows you to call some JavaScript code when the user clicks on it. The `IlvToolBar` class is a component that you can use to arrange `IlvButtons` vertically or horizontally. You will now add some buttons to the page to zoom in and out on the chart. The steps are as follows:

- ◆ *Importing the JavaScript files*
- ◆ *Creating the toolbar*
- ◆ *Creating buttons*
- ◆ *Updating the layoutPage function*

Importing the JavaScript files

First, you must include the JavaScript files that define the `IlvButton` and `IlvToolBar` classes. The JavaScript import statements now look like this:

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



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

Creating the toolbar

In the page body, you first create the vertical toolbar and give it the same background image as the main background panel:

```
var backgroundPattern = ilvImagePath + 'skybg.jpg';
var toolbar = new IlvToolBar(0, 0);
toolbar.setOrientation(IlvToolBar.VERTICAL);
toolbar.setBackgroundColor('#53537A');
toolbar.setBackgroundImage(backgroundPattern);
```

Notice that the initial position of the toolbar is set to (0, 0) and its `toHTML` function is not called until the `layoutPage` function has calculated the correct position of the toolbar and generated its HTML code. This approach makes the page more maintainable by encapsulating all the component positioning into the `layoutPage` function and eliminates complex initial position calculations when you create each component.

Creating buttons

Next, you create three buttons and add them to the toolbar. Each button is defined by its position, size, three images, and a piece of JavaScript to be executed when the button is clicked. The actual positioning of each button will be controlled by the toolbar it is contained in, so you simply set the initial position of each button to (0, 0). The three images used by the button are:

- ◆ The main image specified in the `IlvButton` constructor. This is the image used when the mouse is not over the button and the button is not selected.
- ◆ The rollover image is used when the mouse is over the button, but the button is not selected.
- ◆ The selected image is used when the mouse button is pressed on the button.

The code to create the three zoom buttons is:

```
// Create the Zoom-In toolbar button.
var zoomInAction = function() {
    chartView.zoomIn();
};
var zoomInButton = new IlvButton(0, 0, 20, 20,
                                ilvImagePath+'zoomin-up.gif',
                                zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
```

```

var zoomOutAction = function() {
    chartView.zoomOut();
};
var zoomOutButton = new IlvButton(0, 0, 20, 20,
    ilvImagePath+'zoomout-up.gif',
    zoomOutAction);
zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
zoomOutButton.setToolTipText("Zoom Out");
zoomOutButton.setMessage("Press to zoom out");
zoomOutButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomOutButton);

// Create the Zoom-To-Fit toolbar button.
var zoomFitAction = function() {
    chartView.zoomToFit();
};
var zoomFitButton = new IlvButton(0, 0, 20, 20,
    ilvImagePath+'zoomfit-up.gif',
    zoomFitAction);
zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
zoomFitButton.setToolTipText("Zoom To Fit");
zoomFitButton.setMessage("Press to zoom to fit");
zoomFitButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomFitButton);

```

The `zoomInButton` simply calls the `zoomIn` method of the `IlvGanttView`, the `zoomOutButton` calls the `zoomOut` method, and the `zoomFitButton` calls the `zoomToFit` method. Each button also 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 `IlvHTMLPanel` positioned on the page. This is accomplished by setting the `messagePanel` property of the buttons.

Updating the `layoutPage` function

Finally, you must update the `layoutPage` function to arrange the toolbar on the page and generate the HTML code for the toolbar:

```

var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
        chart.getTop()+chart.getHeight()+15,
        chart.getWidth()-logoPanel.getWidth()-15,
        logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
        chart.getTop() - 10,
        chart.getWidth()+ 20,
        messagePanel.getTop()
            + messagePanel.getHeight()
            + 20
            - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
        + messagePanel.getWidth()
        + 15,

```

```

        messagePanel.getTop());
    toolbar.setLocation(backgroundPanel.getLeft()
        + backgroundPanel.getWidth()
        + 15,
        backgroundPanel.getTop());
}
layoutPage(chartView);
chartView.addSizeListener(layoutPage);
toolbar.toHTML();

```

The complete body of the page is now:

```

<body onload="init()" onunload="ilvDispose()"
    onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
    // The Gantt chart servlet.
    var servletName = "/gantt/GanttChartServlet";
    // The background image.
    var backgroundPattern = ilvImagePath + 'skybg.jpg';

    // Position of the Gantt chart.
    var chartX = 25;
    var chartY = 25;
    var chartH = 350;
    var chartW = 700;

    var backgroundPanel = new IlvHTMLPanel('');
    backgroundPanel.setBackgroundImage(backgroundPattern);
    backgroundPanel.setBackgroundColor('#909090');
    backgroundPanel.setVisible(true);

    var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
    chartView.setServletURL(servletName);
    chartView.setResizable(true);
    chartView.toHTML();

    var messagePanel = new IlvHTMLPanel('');
    messagePanel.setBackgroundColor('#B6D5DA');
    messagePanel.setVisible(true);
    chartView.setMessagePanel(messagePanel);

    var logoPanel = new IlvImageView(0, 0, 67, 30,
        ilvImagePath+'ilog-small.gif');
    logoPanel.toHTML();

    var toolbar = new IlvToolBar(0, 0);
    toolbar.setOrientation(IlvToolBar.VERTICAL);
    toolbar.setBackgroundColor('#53537A');
    toolbar.setBackgroundImage(backgroundPattern);

    // Create the Zoom-In toolbar button.
    var zoomInAction = function() {
        chartView.zoomIn();
    }

```

```

};
var zoomInButton = new IlvButton(0, 0, 20, 20,
                                ilvImagePath+'zoomin-up.gif',
                                zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
var zoomOutAction = function() {
    chartView.zoomOut();
};
var zoomOutButton = new IlvButton(0, 0, 20, 20,
                                ilvImagePath+'zoomout-up.gif',
                                zoomOutAction);
zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
zoomOutButton.setToolTipText("Zoom Out");
zoomOutButton.setMessage("Press to zoom out");
zoomOutButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomOutButton);

// Create the Zoom-To-Fit toolbar button.
var zoomFitAction = function() {
    chartView.zoomToFit();
};
var zoomFitButton = new IlvButton(0, 0, 20, 20,
                                ilvImagePath+'zoomfit-up.gif',
                                zoomFitAction);
zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
zoomFitButton.setToolTipText("Zoom To Fit");
zoomFitButton.setMessage("Press to zoom to fit");
zoomFitButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomFitButton);

var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop()+chart.getHeight()+15,
                           chart.getWidth()-logoPanel.getWidth()-15,
                           logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft()-10,
                              chart.getTop()-10,
                              chart.getWidth()+20,
                              messagePanel.getTop()
                                + messagePanel.getHeight()
                                + 20
                              - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
                          + messagePanel.getWidth()
                          + 15,

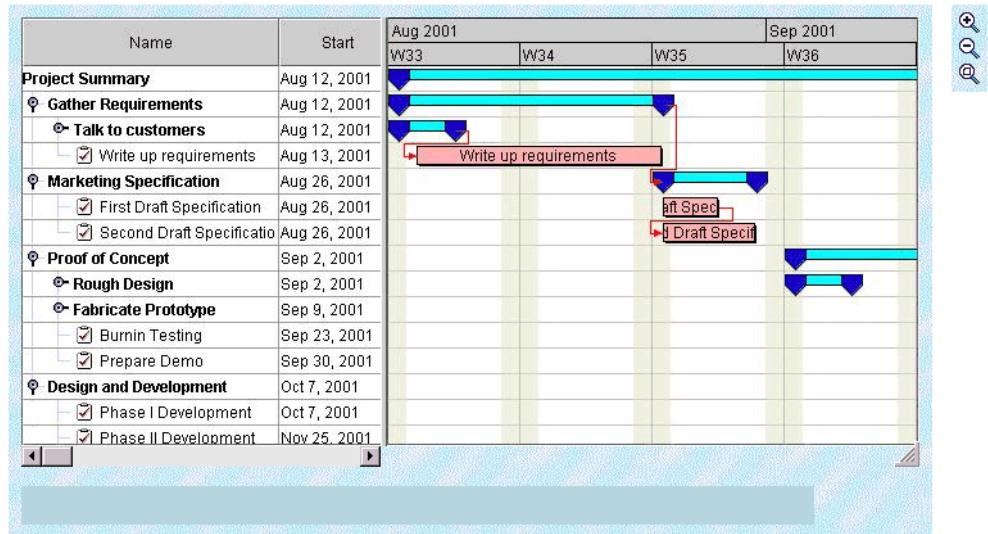
```

```

        messagePanel.getTop());
        toolbar.setLocation(backgroundPanel.getLeft()
            + backgroundPanel.getWidth()
            + 15,
            backgroundPanel.getTop());
    }
    layoutPage(chartView);
    chartView.addSizeListener(layoutPage);
    toolbar.toHTML();
</script>
</body>

```

The page now looks like this with our new vertical toolbar:



IlvGanttSheetScrollInteractor

Until now, you have added components and buttons to the page. You will now add an interactor that allows direct interaction with the image. The `IlvGanttSheetScrollInteractor` allows the user to interactively scroll and pan the image of the Gantt sheet. The `IlvGanttView` component is composed of two child views: an `IlvGanttTableView` that displays the image of the table on the left side of the splitter, and an `IlvGanttSheetView` that displays the image of the Gantt sheet on the right side of the splitter. This is shown in *Gantt DHTML Components*. You set interactors separately on the Gantt table and sheet views. The code to create an `IlvGanttSheetScrollInteractor` and set it on the `IlvGanttSheetView` is very simple:

```

var sheetView = chartView.getSheetView();
var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
sheetView.setInteractor(sheetScrollInteractor);

```

In order to use the `IlvGanttSheetScrollInteractor` class, you must include the JavaScript files that define the class and its base class, `IlvInteractor`:

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

IlvRowExpandCollapseInteractor

The `IlvRowExpandCollapseInteractor` can be set on both an `IlvGanttTableView` and an `IlvGanttSheetView`. It allows the user to click on rows in the image to expand and collapse them. You will set the interactor on the Gantt table view. The steps are as follows:

1. Include the JavaScript file that defines the class:

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

2. Set the interactor on the `IlvGanttTableView`:

```
var tableView = chartView.getTableView();
var tableToggleRowInteractor =
    new IlvRowExpandCollapseInteractor('toggleRow');
tableView.setInteractor(tableToggleRowInteractor);
```

The string `'toggleRow'` refers to a named action on the server side that this interactor invokes. On the server side, this action is registered with the servlet support object in the file:

```
<installdir>/jviews-gantt86/samples/servlet/src/GanttChartServlet.java
```

in the `createServletSupport` method:

```
protected IlvGanttServletSupport createServletSupport()
{
    IlvGanttServletSupport support = new ServletSupport();
    support.addServerAction("toggleRow", new IlvRowExpandCollapseAction());
    return support;
}
```

These types of interactors and how to use them are described in detail in *Adding client/server interactions*.

IlvInteractorButton

The `IlvInteractorButton` class is a subclass of `IlvButton` that installs an interactor on a view. If multiple interactor buttons are defined for the same view, they will behave like radio buttons. When an interactor button is pressed, it installs its interactor and remains pressed until another button installs a different interactor.

In this example, you will define an `IlvRowExpandCollapseInteractor` for the Gantt sheet view. You will then add two interactor buttons to the toolbar that toggle between the scroll interactor and the new row interactor. The steps are as follows:

1. Include the JavaScript file that defines the `IlvInteractorButton` class:

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

2. Create the new interactor for the Gantt sheet view:

```
var sheetView = chartView.getSheetView();
var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
sheetView.setInteractor(sheetScrollInteractor);
var sheetToggleRowInteractor =
    new IlvRowExpandCollapseInteractor('toggleRow');
```

3. Create the interactor buttons and add them to the toolbar:

```
// Create the scroll toolbar button for the Gantt sheet.
var sheetPanButton =
    new IlvInteractorButton(0, 0, 20, 20,
        ilvImagePath+'move-up.gif',
        sheetScrollInteractor,
        sheetView);
sheetPanButton.setRolloverImage(ilvImagePath+'move-sel.gif');
sheetPanButton.setSelectedImage(ilvImagePath+'move-dn.gif');
sheetPanButton.setToolTipText("Pan Gantt Sheet");
sheetPanButton.setMessage("Scroll and pan the Gantt sheet");
sheetPanButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetPanButton);

// Create the toggle row toolbar button for the Gantt sheet.
var sheetToggleRowButton =
    new IlvInteractorButton(0, 0, 20, 20,
        ilvImagePath+'bluearrow-plusminus-up.gif',
        sheetToggleRowInteractor,
        sheetView);
sheetToggleRowButton.setRolloverImage(ilvImagePath+'bluearrow-
    plusminus-sel.gif');
;
sheetToggleRowButton.setSelectedImage(ilvImagePath+'bluearrow-
    plusminus-dn.gif');
sheetToggleRowButton.setToolTipText("Expand/Collapse Gantt Sheet Rows");
sheetToggleRowButton.setMessage("Expand/collapse rows in the Gantt sheet");
;
sheetToggleRowButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetToggleRowButton);
```

The body of the page is now:

```

<body onload="init()" onunload="ilvDispose()"
    onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
    // The Gantt chart servlet.
    var servletName = "/gantt/GanttChartServlet";
    // The background image.
    var backgroundPattern = ilvImagePath + 'skybg.jpg';

    // Position of the Gantt chart.
    var chartX = 25;
    var chartY = 25;
    var chartH = 350;
    var chartW = 700;

    var backgroundPanel = new IlvHTMLPanel('');
    backgroundPanel.setBackgroundImage(backgroundPattern);
    backgroundPanel.setBackgroundColor('#909090');
    backgroundPanel.setVisible(true);

    var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
    chartView.setServletURL(servletName);
    chartView.setResizable(true);
    chartView.toHTML();

    var sheetView = chartView.getSheetView();
    var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
    sheetView.setInteractor(sheetScrollInteractor);
    var sheetToggleRowInteractor =
        new IlvRowExpandCollapseInteractor('toggleRow');

    var tableView = chartView.getTableView();
    var tableToggleRowInteractor =
        new IlvRowExpandCollapseInteractor('toggleRow');
    tableView.setInteractor(tableToggleRowInteractor);

    var messagePanel = new IlvHTMLPanel('');
    messagePanel.setBackgroundColor('#B6D5DA');
    messagePanel.setVisible(true);
    chartView.setMessagePanel(messagePanel);

    var logoPanel = new IlvImageView(0, 0, 67, 30,
                                    ilvImagePath+'ilog-small.gif');
    logoPanel.toHTML();

    var toolbar = new IlvToolBar(0, 0);
    toolbar.setOrientation(IlvToolBar.VERTICAL);
    toolbar.setBackgroundColor('#53537A');
    toolbar.setBackgroundImage(backgroundPattern);

    // Create the scroll toolbar button for the Gantt sheet.
    var sheetPanButton =
        new IlvInteractorButton(0, 0, 20, 20,
                                ilvImagePath+'move-up.gif',

```



```

        sheetScrollInteractor,
        sheetView);
sheetPanButton.setRolloverImage(ilvImagePath+'move-sel.gif');
sheetPanButton.setSelectedImage(ilvImagePath+'move-dn.gif');
sheetPanButton.setToolTipText("Pan Gantt Sheet");
sheetPanButton.setMessage("Scroll and pan the Gantt sheet");
sheetPanButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetPanButton);

// Create the toggle row toolbar button for the Gantt sheet.
var sheetToggleRowButton =
    new IlvInteractorButton(0, 0, 20, 20,
        ilvImagePath+'bluearrow-plusminus-up.gif',
        sheetToggleRowInteractor,
        sheetView);
sheetToggleRowButton.setRolloverImage(ilvImagePath+'bluearrow-
    plusminus-sel.gif');
sheetToggleRowButton.setSelectedImage(ilvImagePath+'bluearrow-
    plusminus-dn.gif');
sheetToggleRowButton.setToolTipText("Expand/Collapse Gantt Sheet Rows");
sheetToggleRowButton.setMessage("Expand/collapse rows in the Gantt sheet");

sheetToggleRowButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetToggleRowButton);

// Create a toolbar separator.
var separator = new IlvButton(0, 0, 20, 10,
    ilvImagePath+'horzsep.gif');
toolbar.addButton(separator);

// Create the Zoom-In toolbar button.
var zoomInAction = function() {
    chartView.zoomIn();
};
var zoomInButton = new IlvButton(0, 0, 20, 20,
    ilvImagePath+'zoomin-up.gif',
    zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
var zoomOutAction = function() {
    chartView.zoomOut();
};
var zoomOutButton = new IlvButton(0, 0, 20, 20,
    ilvImagePath+'zoomout-up.gif',
    zoomOutAction);
zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
zoomOutButton.setToolTipText("Zoom Out");

```

```

zoomOutButton.setMessage("Press to zoom out");
zoomOutButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomOutButton);

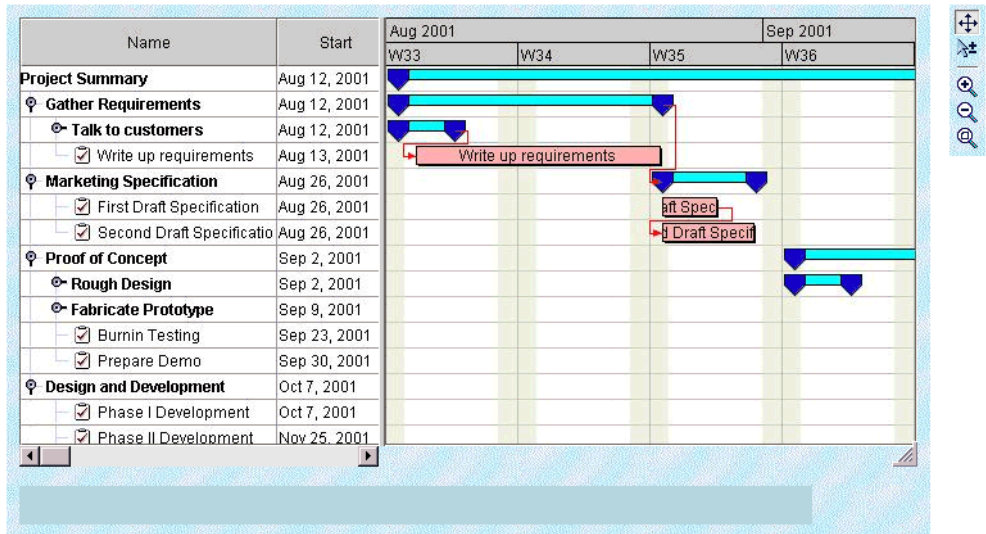
// Create the Zoom-To-Fit toolbar button.
var zoomFitAction = function() {
    chartView.zoomToFit();
};
var zoomFitButton = new IlvButton(0, 0, 20, 20,
    ilvImagePath+'zoomfit-up.gif',
    zoomFitAction);
zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
zoomFitButton.setToolTipText("Zoom To Fit");
zoomFitButton.setMessage("Press to zoom to fit");
zoomFitButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomFitButton);

var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
        chart.getTop()+chart.getHeight()+15,
        chart.getWidth()- logoPanel.getWidth()-15,
        logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
        chart.getTop() - 10,
        chart.getWidth()+ 20,
        messagePanel.getTop()
            + messagePanel.getHeight()
            + 20
            - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
        + messagePanel.getWidth()
        + 15,
        messagePanel.getTop());
    toolbar.setLocation(backgroundPanel.getLeft()
        + backgroundPanel.getWidth()
        + 15,
        backgroundPanel.getTop());
}
layoutPage(chartView);
chartView.addSizeListener(layoutPage);

toolbar.toHTML();
</script>
</body>

```

This results in the following page:



You can now click the first two toolbar buttons to select which interactor is installed on the Gantt sheet view.

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 component is `IlvGanttPopupMenu`.

```
<script TYPE="text/javascript" src="script/IlvAbstractPopupMenu.js"></script>
<script TYPE="text/javascript" src="script/gantt/IlvGanttPopupMenu.js">
</script>
```

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.

Defining a static popup menu in the HTML file

```
//Creates the popup menu.
var popupmenu = new IlvGanttPopupMenu(true);

//Creates the menu model.
var root = new IlvMenu("root");
var item1 = new IlvMenuItem("item1", true, "alert('item1 clicked')");
var item2 = new IlvMenuItem("item2", true, "alert('item2 clicked')");
root.add(item1);
root.add(item2);

//Sets the menu model to the popup menu.
popupMenu.setMenu(root);

[...]

//Sets the popup menu to the view.
chartView.setPopupMenu(popupMenu);
```

Configuring servlet support for a popup menu

```
public class GanttChartServlet extends IlvGanttServlet
{
    [...]
    protected void configureServletSupport(IlvGanttServletSupport support) {
        [...]
        support.setPopupMenuEnabled(true);
    }
}
```

```
[...]  
}
```

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.

Declaring a contextual popup menu and setting it on the view, client side

```
var popupMenu = new IlvGanttPopupMenu(true);  
  
//Sets the popup menu to the view  
chartView.setPopupMenu(popupMenu);
```

On the server side, you need to configure the servlet support to handle popup menus and to set the factory that will generate the menu.

Configuring servlet support and setting the factory, server side

```
public class GanttChartServlet extends IlvGanttServlet {  
  
    [...]  
    protected void configureServletSupport (IlvGanttServletSupport support) {  
        [...]  
            support.setPopupMenuEnabled(true);  
            support.getPopupMenuSupport().setMenuFactory(new SimpleMenuFactory());  
        }  
        [...]  
    }  
}
```

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.

```
[...]
```

```
<style>
.PopupMenuItem {
    background: #21bdbd;
    color: black;
    font-family: sans-serif;
    font-size: 12px;
}

.PopupMenuItemHighlighted {
    background: #057879;
    font-style: italic;
    color: white;
}

.PopupMenuItemDisabled {
    background-color: #EEEEEE;
    font-style: italic;
    color: black;
}
</style>

[...]

<script>

var popupMenu = new IlvGanttPopupMenu(true);
popupMenu.setItemStyleClass('PopupMenuItem');
popupMenu.setItemHighlightedStyleClass('PopupMenuItemHighlighted');
popupMenu.setItemDisabledStyleClass('PopupMenuItemDisabled');

</script>
```

Adding client/server interactions

The JViews Gantt 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 name of an activity that appears on the generated image. Part of this action, clicking the image to select the activity, must be done on the client side. Changing the name of the activity in the Gantt data model 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.

In this section

The client side

Explains how to use the `performAction` method.

The server side

Explains how to detect and execute an action request.

Actions that modify chart capabilities

Explains how to use state information to manage user preferences.

The client side

In a dynamic HTML client, you tell the server to perform an action using the `performAction` method of the `IlvGanttTableView` or `IlvGanttSheetView` JavaScript™ component. Here is an example that asks the server side to execute the action “setName” with coordinate and string parameters, assuming that `view` is an `IlvGanttSheetView`:

```
var x = 100;
var y = 50;
var params = new Array();
params[0]=x;
params[1]=y;
params[2]="New Activity Name";
view.performAction("setName", 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: Section *Actions that modify chart capabilities* discusses server actions that require two client/server round trips.

Creating a custom interactor

This section explores how to create a custom client-side interactor that will allow you to click on an activity graphic in the `IlvGanttSheetView` and ask the server side to execute the “setName” action. You start by defining a new `ActivityNameInteractor` class, then you override the `mouseDown` method. Finally, you make your interactor safer by overriding the `setView` method.

Defining a new interactor class

First, you define the new `ActivityNameInteractor` class as a subclass of `IlvInteractor`. `IlvInteractor` is the base class for all client-side interactors that operate on JavaScript view components:

```
function ActivityNameInteractor() {
    this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");
```

Overriding the mouseDown method

Then, you override the `mouseDown` method to convert the mouse coordinates to be relative to the Gantt sheet and request the server side to perform the “setName” action:


```

function ActivityNameInteractor() {
    this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");

ActivityNameInteractor.prototype.mouseDown = function(e) {
    // The JavaScript view component is always stored in the view
    // instance variable of the interactor.
    var view = this.view;
    // The Y position of the mouse event is relative to the top of the DHTML
    // IlvGanttSheetView. The action needs a Y position relative to the top of
    // the Gantt sheet, ignoring the time scale.
    var actionYPos = e.mouseY - view.cap_tableHeaderHeight;
    if (actionYPos < 0) // Mouse is in timescale, so ignore
        return;
    // Create parameters for the setName action and send it to the server side.

    var params = new Array();
    params[0]=e.mouseX;
    params[1]=actionYPos;
    params[2]="New Activity Name";
    view.performAction("setName", params);
}

```

Notice how the `cap_tableHeaderHeight` instance variable of the `IlvGanttSheetView` is used to subtract out the height of the time scale. This variable is one of several that are initialized when the server side sends capabilities information to the view. The full details of the capabilities request are described in *The capabilities request*. You can find details on the other instance variables that the view initializes from the capabilities information in the *JavaScript Reference Manual* for the `IlvGanttComponentView` method.

`IlvGanttComponentView` is the superclass of `IlvGanttSheetView`.

Making the interactor safe

You can make our new interactor a little bit safer to use by overriding the `setView` method. This method is inherited from `IlvInteractor` and is invoked automatically when an interactor is set on or removed from a view.

By overriding this method, you can verify that the view is indeed an instance of `IlvGanttSheetView`:

```

function ActivityNameInteractor() {
    this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");

ActivityNameInteractor.prototype.mouseDown = function(e) {
    // The JavaScript view component is always stored in the view
    // instance variable of the interactor.

```

```

var view = this.view;
// The Y position of the mouse event is relative to the top of the DHTML
// IlvGanttSheetView. The action needs a Y position relative to the top of
// the Gantt sheet, ignoring the time scale.
var actionYPos = e.mouseY - view.cap_tableHeaderHeight;
if (actionYPos < 0) // Mouse is in timescale
    return;
// Create parameters for the setName action and send it to the server side.

var params = new Array();
params[0]=e.mouseX;
params[1]=actionYPos;
params[2]="New Activity Name";
view.performAction("setName", params);
}

ActivityNameInteractor.prototype.setView = function(view) {
    if (view != null && !view instanceof IlvGanttSheetView) {
        alert("ActivityNameInteractor can only be set on an IlvGanttSheetView");
    }
}
}

```

The server side

On the server side, you need to detect that an action was requested and execute the action before the image is generated and sent back to the client. This is done by implementing the `IlvServerAction` 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 `IlvGanttServletSupport` using the `addServerAction(java.lang.String, ilog.views.gantt.servlet.IlvServerAction)` method.

For the “setName” action, you would add the following lines of code in the `createServletSupport` method of the example `GanttChartServlet`:

```
protected IlvGanttServletSupport createServletSupport()
{
    IlvGanttServletSupport support = new ServletSupport();
    support.addServerAction("setName", new IlvServerAction()
    {
        public void actionPerformed(ServerActionEvent event)
            throws ServletException;
        {
            int x = event.getIntParameter(0);
            int y = event.getIntParameter(1);
            String name = event.getStringParameter(2);
            IlvHierarchyChart chart = event.getChart();
            IlvGraphic graphic = chart.getGanttSheet().getGraphic(new Point(x, y)
    );
            if (graphic instanceof IlvActivityGraphic) {
                IlvActivity activity = ((IlvActivityGraphic)graphic).getActivity();

                activity.setName(name);
            }
        }
    });
    return support;
}
}});
```

Actions that modify chart capabilities

The DHTML client maintains certain state information about the server-side `IlvHierarchyChart` object that it is displaying. You call this basic set of state information *capabilities*. The capabilities information is sent by the server to the client when the client side requests it. When the client side requests an updated set of capabilities data, the server also sends an updated image to the client. The capabilities data includes the number of rows currently visible, the height of the rows, and other basic state information that allows the client to intelligently scroll and manipulate the chart images. The full details of the capabilities request and chart data are described in section *The capabilities request*.

Some server actions requested by the client may modify the capabilities state information for the chart. In this case, the client must be able to request that the server perform the action, send updated capabilities information to the client, and then send an updated image to the client. For example, suppose you want to allow the user to click on a row in the table and toggle the expand/collapse state of the row. As in the previous example, toggling the row must be performed on the server side. However, expanding and collapsing rows in the server-side chart modifies the capabilities data on how many rows are visible. The client side must be updated with the new capabilities so that client-side scrolling and row hit testing can be performed correctly.

You again use the `performAction` method of the `IlvGanttTableView` or `IlvGanttSheetView` DHTML components to request this type of server action. This time however, you set the optional third parameter, `updateAll`, to `true`. This requests the server to send updated capabilities to the client, in addition to performing the action and sending an updated image. Here is some example JavaScript™ code that asks the server side to execute the action “toggleRow” with a y-coordinate relative to the first row in the table, assuming that `view` is an `IlvGanttTableView`:

```
var mouseY = .. mouse pos relative to top of IlvGanttTableView ..
// Take table header into account.
mouseY = mouseY - view.cap_tableHeaderHeight;
// If mouse is in table header, nothing to do
if (mouseY < 0)
    return;
// Take vertical scroll position into account.
mouseY = mouseY + view.getVerticalScrollPosition();
var params = new Array();
params[0] = mouseY;
view.performAction("toggleRow", params, true);
```

As in the previous example, you register the action on the server side using the `addServerAction(java.lang.String, ilog.views.gantt.servlet.IlvServerAction).method`. For the “toggleRow” action, you would add the following lines of code in the `createServletSupport` method of the example `GanttChartServlet`:

```
protected IlvGanttServletSupport createServletSupport()
{
    IlvGanttServletSupport support = new ServletSupport();
    support.addServerAction("toggleRow", new IlvServerAction()
    {
        public void actionPerformed(ServerActionEvent event)
```

```
throws ServletException;
{
  int yPos = event.getIntParameter(0);
  IlvHierarchyChart chart = event.getChart();
  IlvHierarchyNode row = chart.getVisibleRowAtPosition(yPos);
  if (row == null)
    return;
  if (chart.isRowExpanded(row))
    chart.collapseRow(row);
  else
    chart.expandRow(row);
}
});
return support;
}
}});
```


The `IlvGanttServlet` and `IlvGanttServletSupport` classes

Describes how to create a servlet and how the servlet responds to different requests.

In this section

Creating a servlet

Explains how to create a servlet that can produce an image and send it to the client.

The servlet parameters

Explains how the servlet can respond to two different types of HTTP requests.

Multiple sessions

Explains how to create a chart and/or a data model and store them as parameters of a HTTP session.

Creating a servlet

The server side of a thin-client JViews Gantt Web application consists in creating a servlet that can produce an image and send it to the client. The JViews Gantt thin-client support provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvGanttServlet`. This class, which can be found in the package `ilog.views.gantt.servlet`, is an abstract subclass of the `HttpServlet` class from the Java™ servlet API.

Using the `IlvGanttServlet` class is an easy way to create a servlet, but it has one main drawback. You cannot use it to add support for the JViews Gantt thin-client protocol to an existing servlet. This is the purpose of the `IlvGanttServletSupport` class. The `IlvGanttServletSupport` class implements all the JViews Gantt thin-client server-side functionality. In fact, the `IlvGanttServlet` class is just a basic wrapper around an instance of `IlvGanttServletSupport`. The `doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)` method of `IlvGanttServlet` simply calls the `handleRequest` method of its `IlvGanttServletSupport` instance.

In the same way, you can integrate an instance of `IlvGanttServletSupport` into your own servlet to handle the requests coming from the Gantt client side. In our Gantt Servlet example, the code of the servlet can be rewritten using the `IlvGanttServletSupport` class as follows:

```
import ilog.views.gantt.*;
import ilog.views.gantt.servlet.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GanttChartServlet extends HttpServlet
{
    private IlvGanttServletSupport support;

    public void init(ServletConfig config)
        throws ServletException
    {
        super.init(config);
        support = new ServletSupport();
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        if (!support.handleRequest(request, response))
            throw new ServletException("Unrecognized request");
    }

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



```

}

class ServletSupport extends IlvGanttServletSupport
{
    private IlvHierarchyChart _chart;

    public ServletSupport()
    {
        _chart = createChart();
    }

    private IlvHierarchyChart createChart()
    {
        IlvHierarchyChart chart = new IlvGanttChart();
        IlvGanttModel ganttModel = new SimpleProjectDataModel();
        return chart;
    }

    public IlvHierarchyChart getChart(HttpServletRequest request,
                                     IlvServletRequestParameters params)
        throws ServletException
    {
        return _chart;
    }
}

```

In this code you have created a new servlet class, `GanttChartServlet`, that is derived directly from the `HttpServlet` class. The `doGet` method passes the requests to an instance of the `IlvGanttServletSupport` class for handling.

The servlet parameters

The JViews Gantt servlet support can respond to two different types of HTTP requests, the *image request* and the *capabilities request*. The image request returns an image from the Gantt or Schedule chart. The capabilities request returns information to the client, such as the number of rows visible in the chart, the height of the rows, and the minimum and maximum times for horizontally scrolling the Gantt sheet. This information allows the client to know the capabilities of the chart in order to intelligently scroll and manipulate the chart images. When developing the client side of your application, you will use the DHTML scripts provided by the Gantt thin-client support. The scripts 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 an image from the chart. Here is an example of a request for the image of the table portion of a chart, assuming that `myservlet` is the name of the servlet:

```
http://host/servlets/myservlet?request=image
&comp=table
&width=300
&height=250
```

And here is an example for the image of the Gantt sheet portion of a chart:

```
http://host/servlets/myservlet?request=image
&comp=sheet
&width=500
&height=250
&startTime=2001,0,1
&endTime=2001,5,30
```

The Gantt sheet will display the time period from January 1, 2001 to June 30, 2001. A detailed listing of the image request parameters can be found in the *Java API Reference Manual* for the `IlvGanttServletSupport` class.

The capabilities request

The capabilities request tells the servlet to return the capabilities information about the chart to the client. The capabilities request has the following syntax:

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

The `format` parameter tells the servlet which format should be used to return the capabilities information. Two formats are supported, HTML or Octet stream.

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

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`.

Note: The JViews Gantt thin-client support does not provide a predefined Java applet thin client.

Full details on the capabilities request and the information returned by the server can be found in the *Java API Reference Manual* for the `IlvGanttServletSupport` class. Details on how the client-side DHTML components save and use the capabilities information can be found in the *JavaScript Reference Manual* for the `IlvGanttComponentView.getCapabilities` method.

Multiple sessions

The Gantt Servlet example presented a very simple example that creates a single Gantt chart for the servlet. This means that all calls to the servlet (that is, all clients) are looking at the same chart and the same data model. In some applications, you may want to have a chart and/or a separate data model for each client. In this case, you might use the notion of *HTTP sessions*. You can then create a chart and/or a data model and store them as parameters of the session.

Here you take our Gantt Servlet example and modify it slightly so that each client has its own chart that is viewing a common data model. This way, each user can toggle rows to expand and collapse without affecting the charts viewed by the other clients. You use an instance of the `IlvGanttSessionAttribute` class to store the chart as an attribute of the HTTP session. This class handles the details of properly disposing server-side GUI components when the user session expires. Our updated `IlvGanttServletSupport` implementation now looks like this:

```
class ServletSupport extends IlvGanttServletSupport
{
    private IlvGanttModel _model;

    private IlvHierarchyChart createChart()
    {
        synchronized(this) {
            if (_model == null)
                _model = new SimpleProjectDataModel();
        }
        IlvHierarchyChart chart = new IlvGanttChart();
        chart.setGanttModel(_model);
        chart.getGanttSheet().setVerticalGrid(new WeekendGrid());
        ... more chart customizations ...
        return chart;
    }

    /**
     * Returns the chart used for the specified request.
     * @param request The current HTTP request.
     * @param params The parameters parsed from the request.
     */
    public IlvHierarchyChart getChart(HttpServletRequest request,
                                       IlvServletRequestParameters params)
        throws ServletException
    {
        IlvHierarchyChart chart = null;
        HttpSession session = request.getSession();
        if (session.isNew()) {
            chart = createChart();
            IlvGanttSessionAttribute chartProxy =
                new IlvGanttSessionAttribute(chart);
            session.setAttribute("IlvHierarchyChart", chartProxy);
        } else {
            IlvGanttSessionAttribute chartProxy =
```

```
        (IlvGanttSessionAttribute)session.getAttribute("IlvHierarchyChart");
        if (chartProxy != null)
            chart = chartProxy.getChart();
    }
    if (chart == null)
        throw new ServletException("session problem");
    return chart;
}
}
```

Note: If you store the chart directly as an attribute of the HTTP session, it will not be properly garbage-collected when the session expires. You must wrapper the chart in an instance of `IlvGanttSessionAttribute` to ensure that the chart is properly disposed of.

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

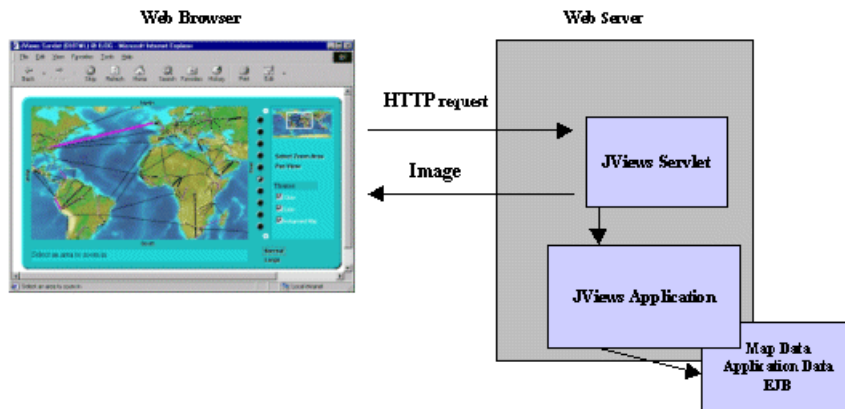
The IBM® ILOG® JViews thin-client support is based on the Java™ servlet technology. Servlets are Java programs that run on a web server. They act as a middle layer between HTTP requests coming from a Web browser or other HTTP clients such as applets or applications and the application or databases on 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.

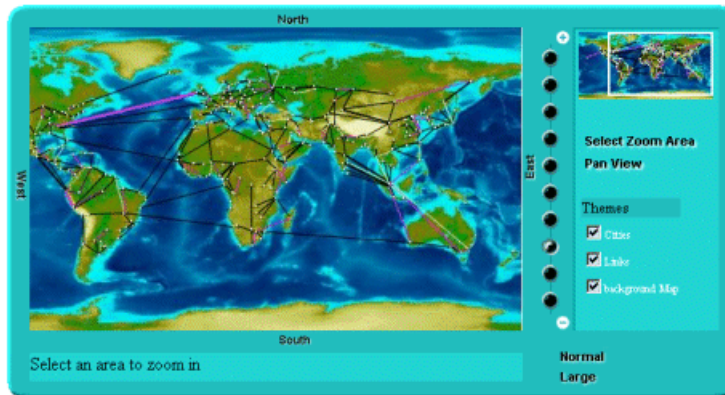
Creating an IBM® ILOG® JViews thin-client application consists of developing the server side and developing the client side.

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 `<install-dir> /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;
}
```

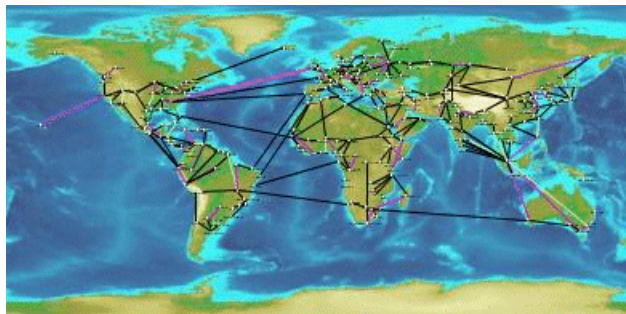
The `getMaxZoomLevel` Method

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 `<install_dir> /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 `<install_dir> /jviews-framework86/samples/xmlgrapher/index.html`.

The full reference documentation of each component can be found in the *JavaScript Reference Manual* located in `<install_dir> /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>
</html>
```

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>
```

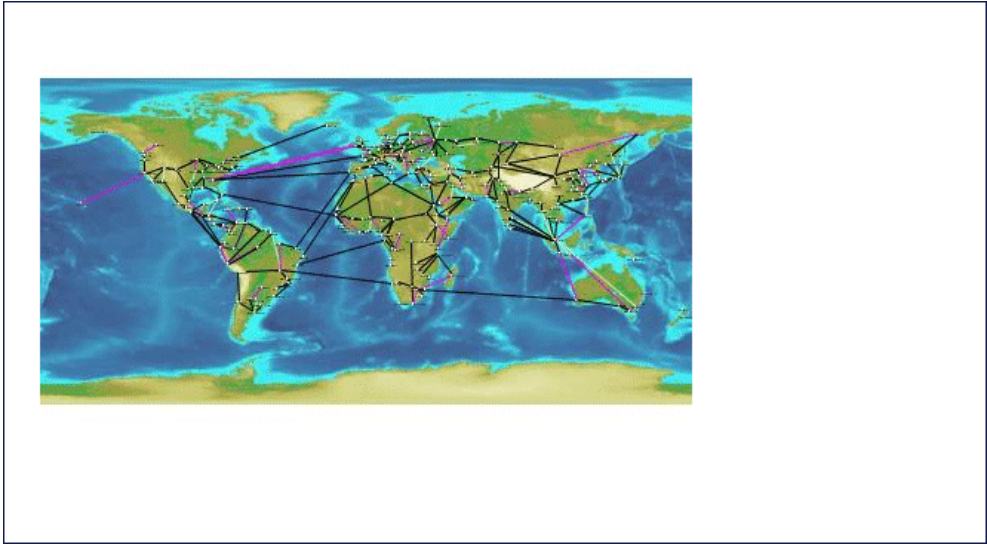
In the body of the page, the example creates an `IlvView` object located in (40, 40) on the HTML page. The size is 440 x 270. This view displays images produced by the servlet `XmlGrapherServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

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>
```

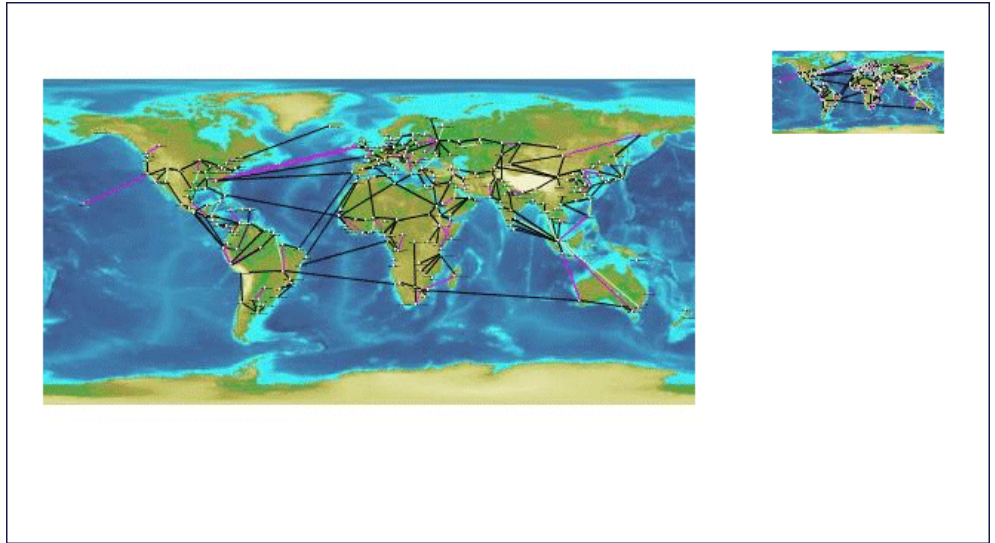
An `IlvOverview` object located in `(x+w+50, y+4)` with a size of `120 x 70` was created:

```
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

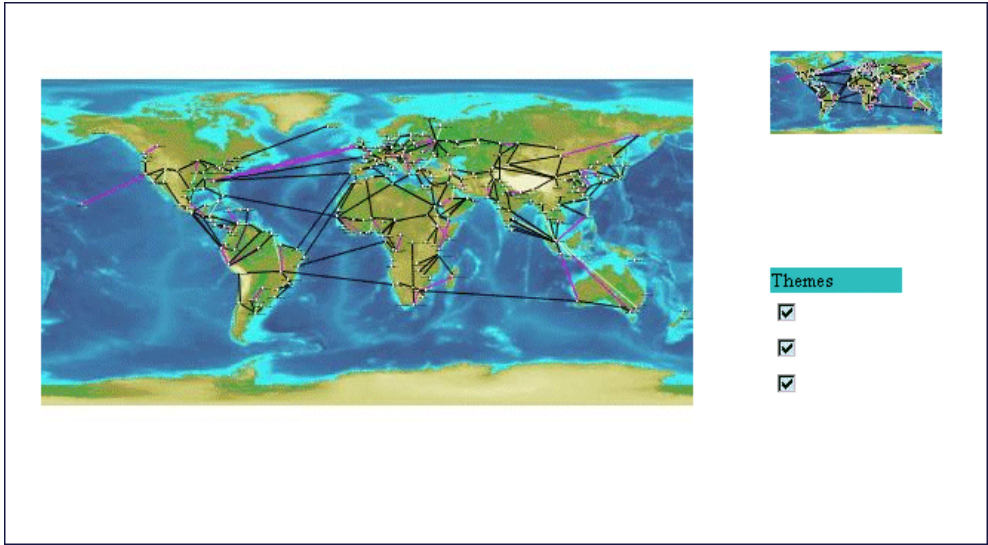
// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')

// 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')

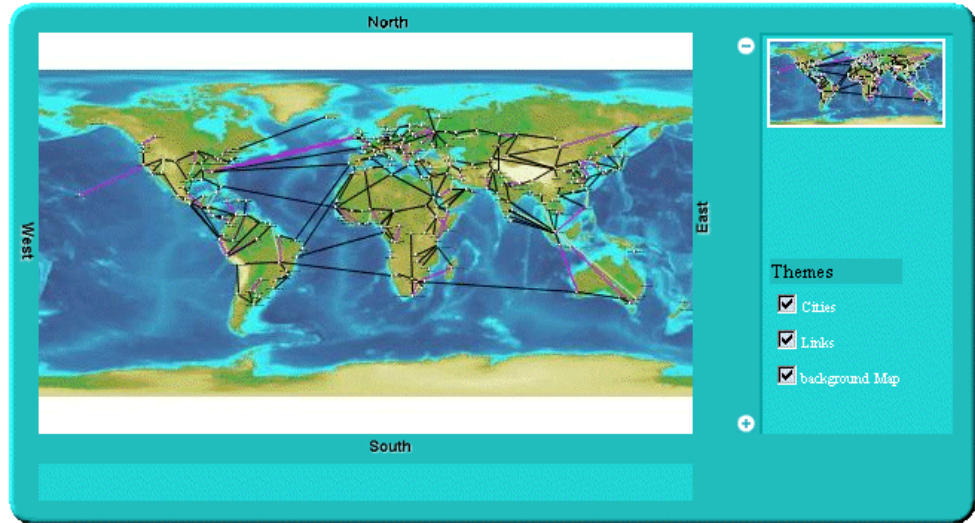
zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif', 'view.
zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')

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

</script>
</body>
</html>

```

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('#21bdbc')

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>
```

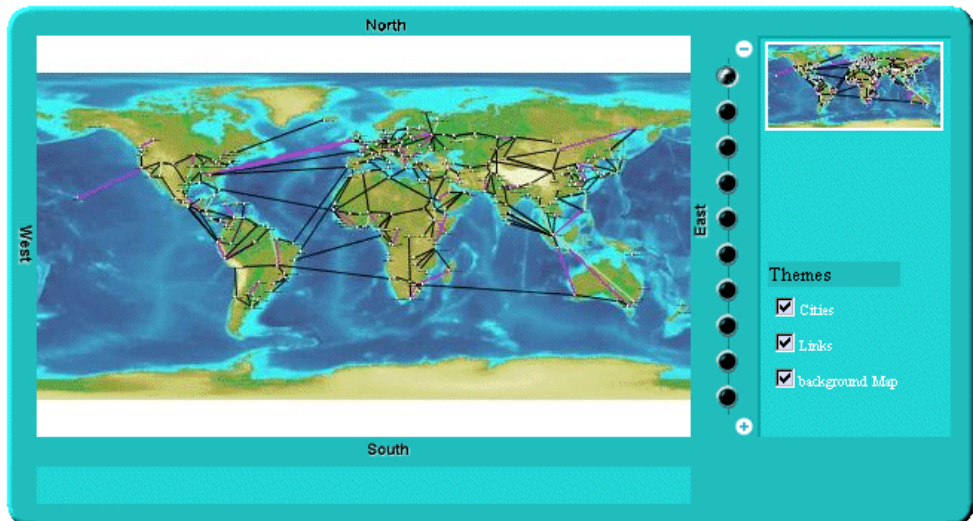
This line imports the script.

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')

...

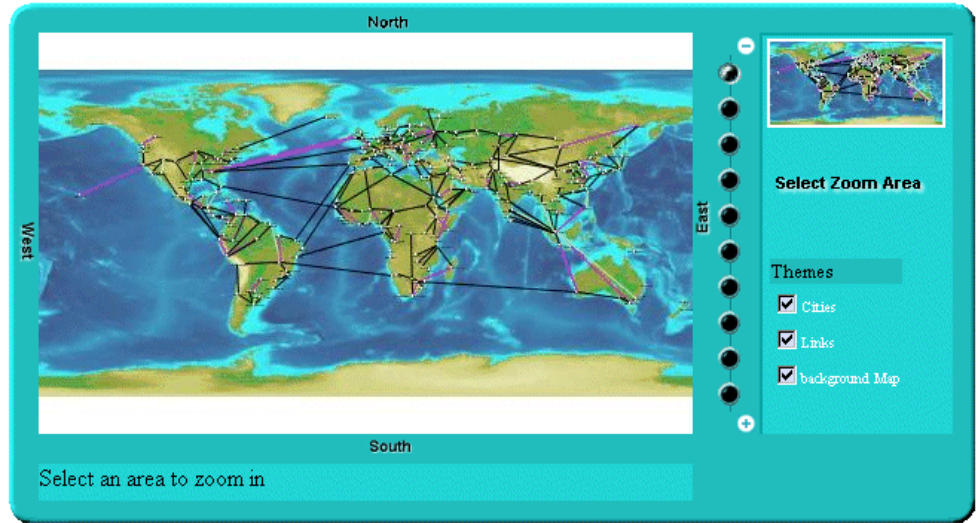
var zoomrectbutton

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.

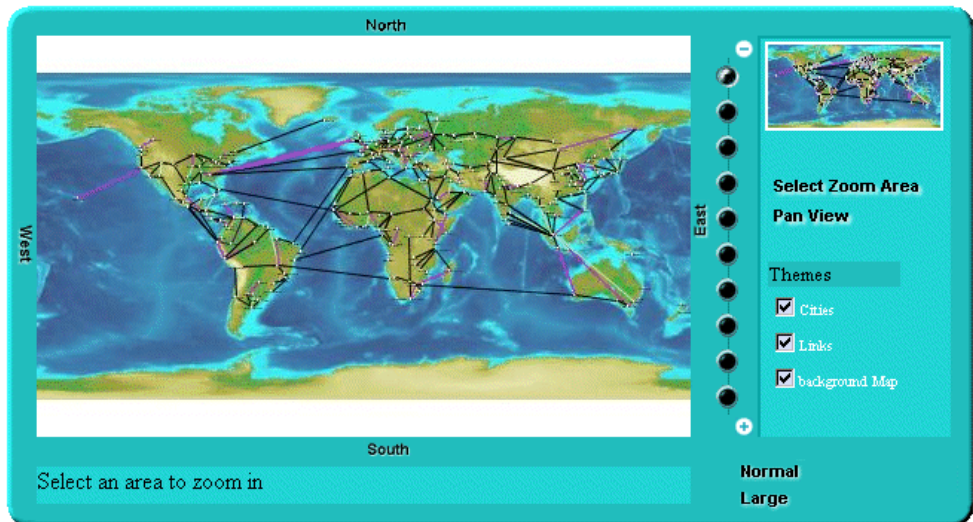
To be able to use the component, import the corresponding JavaScript™ file:

```
<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 component is `IlvGanttPopupMenu`.

```
<script TYPE="text/javascript" src="script/IlvAbstractPopupMenu.js"></script>
<script TYPE="text/javascript" src="script/gantt/IlvGanttPopupMenu.js">
</script>
```

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.

Defining a static popup menu in the HTML file

```
//Creates the popup menu.
var popupmenu = new IlvGanttPopupMenu(true);

//Creates the menu model.
var root = new IlvMenu("root");
var item1 = new IlvMenuItem("item1", true, "alert('item1 clicked')");
var item2 = new IlvMenuItem("item2", true, "alert('item2 clicked')");
root.add(item1);
root.add(item2);

//Sets the menu model to the popup menu.
popupMenu.setMenu(root);

[...]

//Sets the popup menu to the view.
chartView.setPopupMenu(popupMenu);
```

Configuring servlet support for a popup menu

```
public class GanttChartServlet extends IlvGanttServlet
{
    [...]
    protected void configureServletSupport(IlvganttServletSupport support) {
        [...]
        support.setPopupMenuEnabled(true);
    }
}
```

```
[...]  
}
```

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.

Declaring a contextual popup menu and setting it on the view, client side

```
var popupMenu = new IlvGanttPopupMenu(true);  
  
//Sets the popup menu to the view  
chartView.setPopupMenu(popupMenu);
```

On the server side, you need to configure the servlet support to handle popup menus and to set the factory that will generate the menu.

Configuring servlet support and setting the factory, server side

```
public class GanttChartServlet extends IlvGanttServlet {  
  
    [...]  
    protected void configureServletSupport (IlvGanttServletSupport support) {  
        [...]  
        support.setPopupMenuEnabled(true);  
        support.getPopupMenuSupport().setMenuFactory(new SimpleMenuFactory());  
    }  
    [...]  
}
```

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.

```
[...]
```

```
<style>
.PopupMenuItem {
    background: #21bdbd;
    color: black;
    font-family: sans-serif;
    font-size: 12px;
}

.PopupMenuItemHighlighted {
    background: #057879;
    font-style: italic;
    color: white;
}

.PopupMenuItemDisabled {
    background-color: #EEEEEE;
    font-style: italic;
    color: black;
}
</style>

[...]

<script>

var popupMenu = new IlvGanttPopupMenu(true);
popupMenu.setItemStyleClass('PopupMenuItem');
popupMenu.setItemHighlightedStyleClass('PopupMenuItemHighlighted');
popupMenu.setItemDisabledStyleClass('PopupMenuItemDisabled');

</script>
```

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 `IlvViewJavaScript™` 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 `HttpManagerServlet` 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
<code>request</code>	<code>image</code>	Asks the servlet to generate an image.
<code>bbox</code>	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 <code>IlvManager</code> that will be visible.
bgcolor	0xrrggbb For example, 0xffffffff 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>  
  
</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 `IlvManagerServletSupport` 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.

Index

A

- Ajax
 - JavaScript objects for JViews Gantt Faces components **55**
- Ajax-enabled components
 - JSF **18, 25**
- AWT
 - event dispatch thread **96**

C

- capabilities
 - chart capabilities in DHTML thin client **132**
- capabilities request **138**
- client/server interaction
 - adding **127**
 - client side **128**
 - server side **131**
- common DHTML components, overview **100**
- components, servlet and classes **21**
- contextual popup menu
 - dynamic HTML component **125, 178**
 - dynamic HTML component on the client side **125, 178**
 - dynamic HTML component on the server side **125, 178**
 - JSF **51**
 - JSF adding **50**
- CSS
 - styling a data source **36, 37**
- customizing
 - interactor in client-server interaction **128**

D

- dataSourceId attribute **31, 34**
- decorative panels **109**
- deploying
 - as thin client **82**
- deselectAll method

- IlvGanttComponentSelectionManager class **47**
- DHTML components
 - common **100**
 - IlvAbstractPopupMenu **100**
 - IlvAbstractView **100**
 - IlvButton **100, 112**
 - IlvEmptyView **100**
 - IlvGanttComponentView **100**
 - IlvGanttPopupMenu **102**
 - IlvGanttSheetScrollInteractor **102, 117**
 - IlvGanttSheetView **102, 117**
 - IlvGanttTableScrollInteractor **102**
 - IlvGanttTableView **102**
 - IlvGanttView **102, 104**
 - IlvGlassView **100**
 - IlvHTMLPanel **107**
 - IlvImageEventView **100**
 - IlvImageView **100, 109**
 - IlvInteractor **100, 117**
 - IlvInteractorButton **100, 118**
 - IlvMenu **100**
 - IlvMenuItem **100**
 - IlvObject **100**
 - IlvPanel **100**
 - IlvResizableView **100**
 - IlvRowExpandCollapseInteractor **102, 118**
 - IlvScrollBar **100**
 - IlvTableSheetView **117**
 - IlvToolBar **100, 112**
- Dynamic HTML
 - client **100**
- dynamic HTML components
 - contextual popup menu **125, 178**
 - contextual popup menu on the client side **125, 178**

- contextual popup menu on the server side **125, 178**
- IlvMenu **124, 177**
- IlvMenuItem **124, 177**
- prerequisite scripts for popupmenu component **124, 177**
- static popup menu **124, 177**
- dynamic HTML popup menu styling **125, 178**
- dynamic menus **76**

E

- empty view **27**
- event dispatch thread, AWT **96**
- examples

- Gantt Servlet **87**
 - XML Grapher, thin client **85**

F

- Facelets **76**
- faces-config.xml **31, 32, 34, 35**
- files

- GanttChartServlet.java **87**
 - SimpleProjectDataModel.java **87**
 - WAR **89, 104**

G

- Gantt Servlet example
 - description **87**
 - DHTML client **104**
 - installing and running **89**
 - message panel **107**

- gantt-thinclient.war file **89, 104**
- GanttChartServlet.java file **87**
- ganttView
 - JViews Gantt Faces components **25**
- ganttView tag **31**
- getChart method
 - IlvGanttServletSupport class **94**
 - IlvHierarchyChart class **94**
- getDataSource method **31, 34**

H

- handleResize JavaScript function **106**
- HTTP requests. See thin client **83**
- HTTP sessions **140**

I

- IlvAbstractPopupMenu DHTML component **100**
- IlvAbstractView DHTML component **100**
- IlvButton DHTML component **100, 112**
- IlvChart interface **62**
- IlvDataSetPoint class **59**
- IlvDiagrammer interface **62**
- IlvEmptyView DHTML component **100**
- IlvFacesContextualMenu class **21**
- IlvFacesDHTMLGanttChartView class **21**

- IlvFacesDHTMLScheduleChartView class **21**
- IlvFacesGanttSelectInteractor class **21**
- IlvFacesGanttSelectionManager class **21**
- IlvFacesGanttServlet class **21**
- IlvFacesGanttServletSupport class **21**
- IlvFacesNodeSelectInteractor class **21**
- IlvFacesRowExpandCollapseInteractor class **21**
- IlvFacesRowSelectInteractor class **21**
- IlvFacesSheetScrollInteractor class **21**
- IlvFacesTableScrollInteractor class **21**
- IlvGanttChartServlet.ServletSupport inner class **92, 94**
- IlvGanttComponentSelectionManager class
 - deselectAll method **47**
 - selectById method **46**
- IlvGanttComponentView DHTML component **100**
- IlvGanttPopupMenu DHTML component **102**
- IlvGanttSelectionSupport class **46**
- IlvGanttServlet class **92, 97, 135**
- IlvGanttServletSupport class **92, 94, 135**
 - getChart method **94**
- IlvGanttSheetScrollInteractor DHTML component **102, 117**
- IlvGanttSheetView DHTML component **102, 117**
- IlvGanttTableScrollInteractor DHTML component **102**
- IlvGanttTableView DHTML component **102, 117**
- IlvGanttView DHTML component **102, 104**
 - decorative panels **109**
 - resizing **109**
- IlvGanttViewJavaScript class
 - toHTML method **109**
- IlvGlassView DHTML component **100**
- IlvHierarchyChart interface **62**
- IlvHierarchyNode interface **59**
- IlvHTMLPanel DHTML component **107, 109**
- IlvImageEventView DHTML component **100**
- ilvImagePath global variable **109**
- IlvImageView DHTML component **100, 109**
- IlvInteractor class **128**
- IlvInteractor DHTML component **100, 117**
- IlvInteractorButton DHTML component **100, 118**
- IlvManagerView interface **62**
- IlvMenu DHTML component **100**
- IlvMenu dynamic HTML component **124, 177**
- IlvMenuFactory interface **51, 125, 178**
- IlvMenuItem DHTML component **100**
- IlvMenuItem dynamic HTML component **124, 177**
- IlvObject DHTML component **100**
- IlvPanel DHTML component **100**
- IlvResizableView DHTML component **100**
- IlvRowExpandCollapseInteractor DHTML component **102, 118**
- IlvScrollBar DHTML component **100**

IlvSDMNode interface **59**
IlvSelectionPropertiesError
 JavaScript class **42**
IlvServerAction interface **131**
IlvToolBar DHTML component **100, 112**
image request, in client-server interaction **138**
image server
 declaring in portlet mode **69**
image servlet
 interactions **62**
 value change listener **62**
init JavaScript function **106**
interactions
 executing in image servlet context **62**
 executing in JSF lifecycle **59**
interactors, installing **38**
interactors, installing in chart **39**

J

JavaScript
 common DHTML components **88, 100**
 DHTML scripts **83**
 files
 importing **105, 112**
 functions **106**
JavaScript action
 in managed bean **67**
 namespace-encoded variables **67**
 notation **67**
 variables **67**
JavaScript objects **55**
JavaScript variables
 action **67**
 portlet namespace **67**
 javax.servlet.http package **92**
JSF **18**
 components and portlets **67**
JSF components
 integrating into portal **69**
JSF lifecycle
 interactions **59**
 value change listener **59**
JSF menu factory
 contextual popup menu **51**
JSF popup menu
 adding a contextual **50**
 contextual **51**
 contextual menu factory **51**
 static **50**
 styling **53**
JSP **18**
JSR 168
 portlets **67**
jv
 menu tag **50**
 menuItem tag **50**

 menuSeparator tag **50**
JViews Gantt Faces
 Ajax-enabled components **18, 25**
JViews Gantt Faces components
 ganttView **25**

M

managed bean
 JavaScript action **67**
managed-beans.xml **31, 32, 34, 35**
menu binding
 static **76**
menus
 dynamic **76**
message box, connecting chart view **49**
message box, connecting to **48**
message panel **107**
multiple sessions **140**

N

namespace
 JavaScript variables in portlets **67**
 portlet **67**
 scripts in portlets **67**
namespace-encoded variables
 JavaScript action **67**
notation
 JavaScript action **67**

O

octet-stream format for capabilities information
138

P

popup menu
 prerequisite scripts for dynamic HTML
 component **124, 177**
portal
 integrating JSF components **69**
portlets
 and JSF components **67**
 declaring image server **69**
 JSR 168 **67**
 namespace **67**

R

refjavacharts
 ilog/views/chart/data/IlvDataSetPoint.html
 59

S

scripts
 portlet namespace **67**
selectById method
 IlvGanttComponentSelectionManager class
 46
server side of thin client
 developing **91**

- key classes **92**
- multithreading **96**
- servlet
 - capabilities request **138**
 - creating **97**
 - `IlvGanttServlet` class **97**
 - image request **138**
 - multiple sessions **140**
 - parameters **138**
- `setInteractor` method **55**
- simple view **27**
- `SimpleProjectDataModel.java` file **87**
- static menu **76**
- static popup menu
 - dynamic HTML component **124, 177**
 - JSF **50**
- structure of web application **104**
- `styleSheets` attribute **36, 37**
- styling
 - dynamic HTML popup menu **125, 178**
 - JSF popup menu **53**

T

- thin client
 - adding client/server interactions **127**
 - chart capabilities **132**
 - client side, developing **99**
 - DHTML client **100**
 - example **85**
 - server side, developing **91**
 - web architecture **83**
 - XML Grapher example **85**
- `toHTML` method
 - `IlvGanttView JavaScript` class **109**
- Tomcat **89**
- Trinidad **76**

V

- value change listener
 - image servlet **62**
 - JSF lifecycle **59**
- view
 - empty **27**
 - simple **27**

W

- WAR files
 - `gantt-thinclient` **89, 104**
- web application directory structure **104**

X

- XML Grapher, thin client example **85**
- `XMLDataSource` tag **31, 34**