GWT Portlets Framework User Manual

Version 0.9.5beta

19 August 2009

By David Tinker, Jon Lai Lam

1. **Introduction** 1
    1.1. Contents & Usage 2
    1.2. System Requirements 2
    1.3. Demo Applications 2
    1.4. Packages 2
2. **Portlets** 4
    2.1. Hello World Portlet 4
    2.2. Simple CRUD Portlet 6
3. **Other Concepts** 9
    3.1. Containers & Layouts 9
    3.2. WidgetFactory Trees 10
    3.3. Page Files 10
4. **Application Structure** 12
    4.1. Classes 12
    4.2. Event Broadcasting 14
    4.3. The WidgetRefreshHook Singleton 15
    4.4. WidgetFactory Trees and Data Providers 16
5. **Layouts** 17
    5.1. RowLayout 17
    5.2. Using LayoutPanel 17
    5.3. Layout Tips 19
        5.3.1. Scrollbars 19
        5.3.2. Tables 19
        5.3.3. Margins 19
        5.3.4. PositionAware interface 19
    5.4. The LDOM Class 19
6. **The Page Editor** 21
    6.1. Using The Editor 22
7. **UI Widgets** 24
    7.1. Dialog and CssButton 24
    7.2. FormBuilder 25
    7.3. TitlePanel 25
    7.4. WebAppContentPortlet 26
    7.5. MenuPortlet 26
    7.6. PagePortlet 27
    7.7. PageTitlePortlet 27
    7.8. ToolButton 27
    7.9. ShadowPanel 27
8. **Theme Support** 28
    8.1. The Theme Singleton 28
    8.2. Creating Themes 28
9. **Integrating with Spring** 30

Chapter 1. Introduction

GWT Portlets (http://code.google.com/p/gwtportlets/) is a free open source web framework for building GWT (Google Web Toolkit) applications. It defines a very simple & productive, yet powerful programming model to build good looking, modular GWT applications.

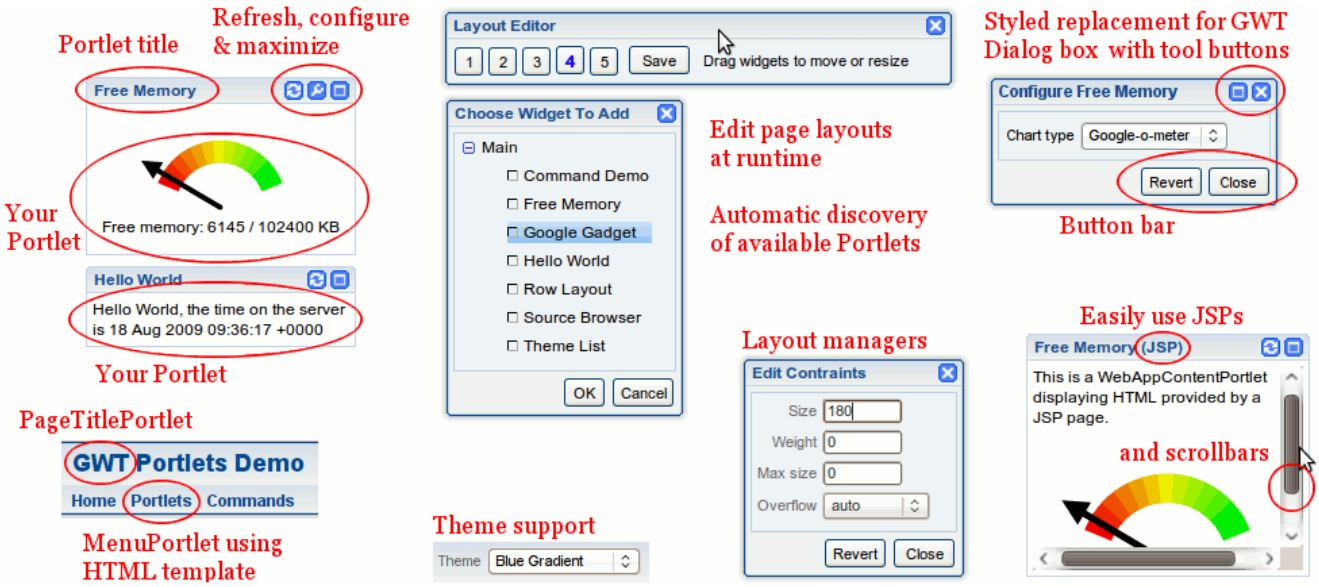


Figure 1.1. Overview Of Functionality

The programming model is somewhat similar to writing JSR168 portlets for a portal server (Liferay, JBoss Portal etc.). The "portal" is your application built using the GWT Portlets framework as a library. Application functionality is developed as loosely coupled Portlets each with an optional server side DataProvider.

Every Portlet knows how to externalize its state into a serializable PortletFactory subclass (momento / DTO / factory pattern) making important functionality possible:

- CRUD operations are handled by a single GWT RPC for all Portlets
- The layout of Portlets on a "page" can be represented as a tree of WidgetFactory's (an interface implemented by PortletFactory)
- Trees of WidgetFactory's can be serialized and marshalled to/from XML on the server, to store GUI layouts (or "pages") in XML page files

Other important features of the framework are listed below:

- Pages can be edited in the browser at runtime (by developers and/or users) using the framework layout editor
- Portlets are positioned absolutely so can use scrolling regions
- Portlets are configurable, indicate when they are busy loading for automatic "loading spinner" display and can be maximized
- Themed widgets including a styled dialog box, a CSS styled button replacement, small toolbuttons and a HTML template driven menu

GWT Portlets is implemented in Java code and does not wrap any external Javascript libraries. It does not impose any server side framework (e.g. Spring or J2EE) but is designed to work well in conjunction with such frameworks.

1.1. Contents & Usage

The download distribution includes binaries, source code, demos and documentation. Everything required to use and build the framework is included. Some files required to build this manual have been excluded to reduce the size of the distribution i.e. the documentation must be built from a subversion checkout.

The framework is packaged as `gwt-portlets.jar` and this file must be on the classpath at compile and runtime (e.g. in WEB-INF/lib). It depends on Log4j (log4j-*.jar) and optionally XStream (xstream-*.jar, xpp3_min-*.jar and xmlpull*.jar). These jars and their licenses are included in the download bundle (`lib` directory).

Add the following line to the GWT module file for your application to use GWT Portlets:

```
<inherits name="org.gwtportlets.portlet.Portlets"/>
```

1.2. System Requirements

The GWT Portlets framework has the following system requirements:

- JDK 1.5 or newer
- Google Web Toolkit 1.7 or newer
- Apache Ant 1.7 or newer is required to run the demo applications

Currently it will work with GWT 1.6.x but our development and testing is done on GWT 1.7.

1.3. Demo Applications

The demos (there is only one at present) are standalone applications with their own Ant build files. Each is in a separate directory under `demos`.

To run the demos you need to:

- Edit `build.properties` to match your environment (set your GWT installation directory etc.)
- Run `ant` (no arguements) at a command prompt from the root of the distribution. This will copy `gwt-portlets.jar`, other required jars and `build.properties` to each of the demos. Each demo will now be standalone (you can copy the whole directory elsewhere to start a project)

Change to `demos/main` and run `ant -p` to list targets. Running `ant` (no arguements) will launch the demo in hosted mode.

1.4. Packages

The framework package layout follows standard GWT conventions:

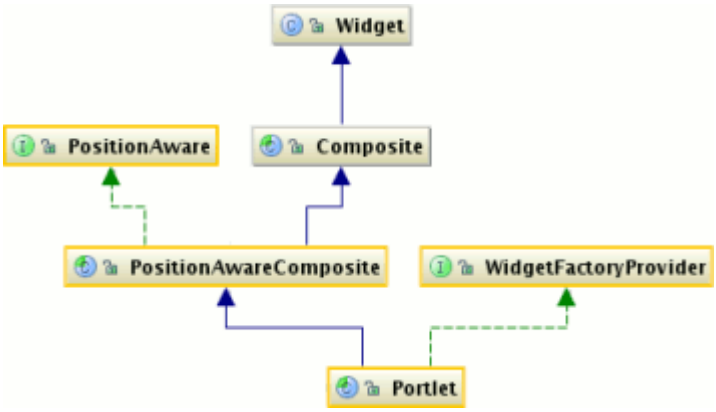
org.gwtportlets.portlet.client	Client side classes that are compiled to Javascript to run in the browser
org.gwtportlets.portlet.client.edit	The in-browser layout editor
org.gwtportlets.portlet.client.event	Application event broadcast support

org.gwtportlets.portlet.client.impl	Browser specific classes and interfaces for compile time generated code
org.gwtportlets.portlet.client.layout	Layout strategies
org.gwtportlets.portlet.client.ui	Widget library
org.gwtportlets.portlet.client.util	Miscelaneous utility classes
org.gwtportlets.portlet.rebind	GWT compile time code generators
org.gwtportlets.portlet.public	Stylesheets and images
org.gwtportlets.portlet.server	Server side support

Chapter 2. Portlets

A portlet is a GWT Widget (it extends Composite) with features that make it easier to build GWT applications composed of decoupled components:

- It can externalize its state into an instance of a PortletFactory subclass that can recreate the Portlet and/or restore its state
- It can "refresh" itself by sending a PortletFactory instance to the server for update (e.g. from a database) and restoring its state using the returned factory to show the new data i.e. the factory is used as a DTO (Data Transfer Object)
- It has a user friendly title
- It may be able open a dialog to configure itself
- It notifies its parent (and its parent's parent recursively) when its state changes (e.g. busy with refresh, title changed) for automatic display of AJAX loading pizza's etc.
- It can be positioned absolutely and is aware of its position and size and hence can use scrolling regions effectively



2.1. Hello World Portlet

This section describes the different parts of the "hello world"portlet from the main demo (demos/main). The Portlet class is as follows:

```
package main.client.ui;

import com.google.gwt.user.client.ui.*;
import org.gwtportlets.portlet.client.*;
import org.gwtportlets.portlet.client.ui.*;

public class HelloWorldPortlet extends Portlet {

    private String serverTime;

    private Label label = new Label();

    public HelloWorldPortlet() {
        initWidget(label);
    }

    private void restore(Factory f) {
```

```
        serverTime = f.serverTime;
        label.setText("Hello World, the time on the server is " + serverTime);
    }

    public WidgetFactory createWidgetFactory() {
        return new Factory(this);
    }

    public static class Factory extends PortletFactory<HelloWorldPortlet> {

        @DoNotSendToServer public String serverTime;

        public Factory() { }

        public Factory(HelloWorldPortlet p) {
            super(p);
            serverTime = p.serverTime;
        }

        public void refresh(HelloWorldPortlet p) {
            super.refresh(p);
            p.restore(this);
        }

        public HelloWorldPortlet createWidget() {
            return new HelloWorldPortlet();
        }
    }
}
```

This portlet displays its message using a single label field. Since it is a Composite it calls initWidget(label) in the constructor.

The static inner class `Factory extends PortletFactory {...}` has a single field String serverTime to hold the state of the portlet. This field is marked with the @DoNotSendToServer annotation which causes the framework to set it to null before the factory is sent to the server, reducing the amount of data transfered.

The portlet method WidgetFactory createWidgetFactory() {...} creates a instance of the factory class externalizing the state of the portlet. This method is specified by the WidgetFactoryProvider interface.

The refresh(HelloWorldPortlet p) {...} method in the factory restores the state of the portlet by calling its restore(Factory f) {...} method. The restore method copies data out of the factory fields into the portlet fields (serverTime) and sets the text on the label to update the GUI.

PortletFactory instances are filled with data on the server side by WidgetDataProvider implementations. Data providers are named after the portlet they supply data for by convention (HelloWorldDataProvider for HelloWorldPortlet etc.). Note that a Portlet is not required to have a corresponding WidgetDataProvider (e.g. the PageTitlePortlet in the framework).

Here is the "hello world" data provider:

```
package main.server;

import main.client.ui.HelloWorldPortlet;
import org.gwtportlets.portlet.server.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorldDataProvider
        implements WidgetDataProvider<HelloWorldPortlet.Factory> {

    private static final SimpleDateFormat DATE_FORMAT =
```

```
            new SimpleDateFormat("dd MMM yyyy HH:mm:ss Z");

    public Class getWidgetFactoryClass() {
        return HelloWorldPortlet.Factory.class;
    }

    public void refresh(HelloWorldPortlet.Factory f, PageRequest req) {
        f.serverTime = DATE_FORMAT.format(new Date());
    }
}
```

The getWidgetFactoryClass() method just returns the factory class that the data provider populates.

The refresh(HelloWorldPortlet.Factory f, PageRequest req) method populates the factory with the time on the server. The PageRequest contains information derived from the current "history token" on the client (the part after the # in the URL). In particular if the history token contains "parameters" then these are available through the PageRequest.

Example: If the history token at the time of refresh is "#hello_world?foo=bar" then req.get("foo") will return "bar". A more complex data provider might use this parameter and information from the factory to execute a database query.

The page request also includes the HttpServletRequest and HttpServletResponse for the call.

2.2. Simple CRUD Portlet

This section describes the different parts of the "simple crud"portlet from the main demo (demos/main). This portlet displays a list of contacts on a grid with buttons to add, edit and delete contacts.

The static inner class Contact is a data transfer object (DTO) which is used to transfer data between the portlet and the data povider.

```
public class SimpleCrudPortlet extends Portlet {
...
    public static class Contact implements Serializable {
      public int contactId;
      public String name;
      public String mobile;
    }
...
}
```

The portlet factory has an array of Contacts which is populated by the data porvider on the server side and displayed by the portlet. This field is annotated `@DoNotSendToServer` to avoid sending this data back to the server when refreshing the Portlet.

The portlet factory also has two other attributes `int deleteContactId` and `Contact update` these are used when deleting, adding and editing a Contact. Note that these fields are annotated `@DoNotPersist` so they are not stored in XML page files.

```
public class SimpleCrudPortlet extends Portlet {
...
    public static class Factory extends PortletFactory<SimpleCrudPortlet> {

      @DoNotSendToServer
      public Contact[] contactList;
      @DoNotPersist
      public int deleteContactId;
```

```
      @DoNotPersist
      public Contact updateContact;
      ...
}
```

The data provider's `refresh(SimpleCRUDPortlet.Factory f, PageRequest req) {...}` populates the contactList field in the factory with a list of contacts. This list is stored in the session on the server to simulate a database.

```
public class SimpleCrudDataProvider
      implements WidgetDataProvider<SimpleCrudPortlet.Factory> {
  ...
  public void refresh(SimpleCrudPortlet.Factory f, PageRequest req) {
      ...
      List<SimpleCrudPortlet.Contact> list = getContacts(req); // get from session
      ...
      f.contactList = list.toArray(new SimpleCrudPortlet.Contact[list.size()]);
      ...
  }
  ...
}
```

The portlet's restore method loops through the ContactList and adds the to the FlexTable.

```
public class SimpleCrudPortlet extends Portlet {
  private FlexTable grid;
  ...
  private void restore(Factory f) {
      contactList = f.contactList;
      ...
      for (int i = 0; i < contactList.length; i++) {
          Contact contact = contactList[i];
          int row = i + 1;
          grid.setText(row, 0, "" + contact.contactId);
          grid.setText(row, 1, contact.name);
          grid.setText(row, 2, contact.mobile);
      }
  }
...
```

Adding and editing a Contact is done by the same dialog. The add method opens the dialog passing a new Contact to the dialog. The edit method gets the selected contact on the grid and passes it to the dialog.

```
private void showContactDialog(final Contact c) {
    final Dialog dlg = new Dialog();
    final TextBox name = new TextBox();
    final TextBox mobile = new TextBox();

    boolean adding = c.contactId == 0;
    dlg.setText((adding ? "Add" : "Edit") + " Contact");
    name.setText(c.name);
    mobile.setText(c.mobile);

    FormBuilder fb = new FormBuilder();
    fb.label("Name").field(name).endRow();
    fb.label("Mobile").field(mobile).endRow();

    dlg.setWidget(fb.getForm());
    dlg.addButton(new CssButton("OK", new ClickHandler() {
        public void onClick(ClickEvent clickEvent) {
            if (name.getText().length() == 0 || mobile.getText().length() == 0) {
                Window.alert("Name and Mobile are required");
                return;
            }
```

```
            Contact dto = new Contact();
            dto.contactId = c.contactId;
            dto.name = name.getText();
            dto.mobile = mobile.getText();
            // send the Contact to be updated to the server via refresh
            Factory f = new Factory(SimpleCrudPortlet.this);
            f.update = dto;
            refresh(f, dlg);
            // dialog hides itself onSuccess and ignores onFailure
            // which is handled by main.client.Demo
        }
    }));
    dlg.addCloseButton("Cancel");
    dlg.showNextTo(grid);
}
```

A new Contact DTO is created using the data from the Dialog when OK is clicked. This is attached to the update field of the Factory and sent to the server via a refresh call. The dialog itself is passed to the refresh method as the AsyncCallback. It hides itself in onSuccess and does nothing in onFailure as errors are handled globally in main.client.Demo.

The SimpleCrudDataProvider handles the refresh as shown below:

```
public class SimpleCrudDataProvider
        implements WidgetDataProvider<SimpleCrudPortlet.Factory> {
    ...
    public void refresh(SimpleCrudPortlet.Factory f, PageRequest req) {
        List<SimpleCrudPortlet.Contact> list = getContacts(req);
        if (f.update != null) {
            update(list, f.update);
        }
        if (f.deleteContactId > 0) {
            delete(list, f.deleteContactId);
        }
        save(req, list);
        f.contactList = list.toArray(new SimpleCrudPortlet.Contact[list.size()]);
    }
    ...
}
```

Deleting a Contact is done when the delete button is clicked, the deleteContactId is set to the selected contact on the grid. The data provider then removes the contact from the list.

The update method in SimpleCrudDataProvider throws an IllegalArgumentException on duplicate names. This causes the refresh to fail and an error alert is displayed on the client side.

Using the Portlet refresh machanism for updates and deletes removes the need for additional RPC methods and associated complexity.

Chapter 3. Other Concepts

This chapter describes the key concepts and ideas used by the framework.

3.1. Containers & Layouts

The framework provides support for absolute positioning of widgets within each other and the browser viewport. Widgets (e.g. Portlets) are arranged using constraints and layout managers in a manner similar to Swing and AWT. This approach makes it possible to create scrolling regions and to build a browser based "desktop style" application. Static (i.e. normal browser flow) positioning can still be used for the "contents" of widgets and is often easier than trying to control the position and size of every button and label using absolute positioning.

The Container interface extends GWTs concept of a "Panel" to add support for pluggable layout managers and layout constraints:

```
package org.gwtportlets.portlet.client.layout;
...
public interface Container extends HasWidgets, IndexedPanel, WidgetFactoryProvider,
          PositionAware {
    ...
    public Layout getLayout();
    public void setLayout(Layout layout);
    public LayoutConstraints getLayoutConstraints(Widget widget);
    public void setLayoutConstraints(Widget widget, LayoutConstraints constraints);

    /**
     * Redo this containers layout. Note that containers do not automatically
     * call layout() when widgets are added/removed etc. Only resizing the
     * container triggers automatic layout.
     */
    public void layout();
    ...
}
```

Note that Container extends PositionAware. PositionAware widgets are notified when their position and/or size may have been updated by a call to the boundsUpdated() method from the interface. Containers redo their layouts and reposition their children in response to a boundsUpdated() call.

Two layouts are included with the framework:

- `RowLayout` Lays out widgets in a row or a column with using minimum sizes and weights to make use of available space. Nested containers using RowLayout can be used to construct "border layout" and others
- `DeckLayout` Places widgets on top of each other. Useful for creating "tab panels" and for putting AJAX loading pizzas above other widgets etc.

LayoutPanel is a general purpose Container implementation used by many of the Widgets in the framework. It is covered in detail on this page and here is a small sample:

```
LayoutPanel panel = new LayoutPanel(); // defaults to RowLayout in a column
panel.add(chart); // use all free space and include scrollbars if needed
panel.add(label, 24); // 24 pixels high, no scrollbars (overflow is hidden)
panel.layout(); // adding widgets does not automatically redo the layout
```

3.2. WidgetFactory Trees

Container extends WidgetFactoryProvider so all container implementations support externalizing their state into a serializable WidgetFactory subclass that can restore that state (like Portlets). A tree of Containers and Widgets implementing WidgetFactoryProvider (e.g. Portlets) can be externalized into a tree of WidgetFactories and recreated with a few lines of code:

```
Container root = ...; // tree of Containers and Portlets

// get tree of WidgetFactories
WidgetFactory wf = root.createWidgetFactory();

// create a copy of the original tree (root)
Widget w = wf.createWidget();
wf.refresh(w);
```

Because WidgetFactories are serializable it is easy to transfer WidgetFactory trees between the client and server. This mechanism is also used to implement undo and redo and other features in the layout editor and to store layouts in XML files.

The WidgetFactory interface also supports the visitor pattern for easy traversal of factory trees (e.g. when populating a tree with data from database on the server).

```
WidgetFactory wf = ...;
// visit is invoked for each factory in the tree
wf.accept(new WidgetFactoryVisitor(){
    public boolean visit(WidgetFactory wf) {...}
});
```

3.3. Page Files

The framework supports the conversion of WidgetFactory trees to/from XML using XStream[1]. Note there is no dependency on XStream i.e. other technologies can be used to persist WidgetFactory trees. However XStream is fast and produces human readable and editable XML with a minimum of configuration.

Here is part of the hello_world page from the demo:

```
<LayoutPanel styleName="" limitMaximize="false">
  <widgets>
    <LayoutPanel styleName="" limitMaximize="false">
      <widgets>
        <LayoutPanel styleName="" limitMaximize="false">
          <widgets>
            <TitlePanel styleName="portlet-title" title="Title" titleAuto="true"
                    refresh="true"
                    configure="true" edit="false" maximize="true" limitMaximize="true">
              <widgets>
                <HelloWorldPortlet styleName=""/>
              </widgets>
              <constraints>
                <RowLayout-Constraints size="0.0" weight="1.0" maxSize="0"
                    overflow="hidden"/>
              </constraints>
...
</LayoutPanel>
```

[1] http://xstream.codehaus.org/

Storing the layout of an application GUI in XML files on the server has several advantages over hardcoding it into the Javascript:

- The application has looser coupling i.e. the EntryPoint class does not have to "know about" all of its GUI components
- All factories on a page can be populated with data in a single page fetch async call
- The layout can be customized for different installations or themes

The XStream support is provided by WidgetFactoryXmlIO. This class uses XStream's alias support to avoid putting fully qualified class names into the XML files. Note that if you rename a Portlet (other than moving it to a new package) you will need to update your page files. Some simple heuristics are used to convert a fully qualified class name into a user friendly alias as described below:

```
package org.gwtportlets.portlet.server;
...
public class WidgetFactoryXmlIO {
...
    /**
     * Add an alias for cls. The alias is the simple name of the class (i.e.
     * without package) with the following modifications:<br>
     * <li>Any 'Factory' suffix is removed
     * <li>Any '$' is replaced with '-' (inner class names)
     * Also omits fields annotated with DoNotPersist or DoNotSendToServer.
     */
    public void alias(Class cls) {...}
...
```

It aliases all the framework classes in its constructor. Applications will usually use Spring or a similar framework to discover all of the PortletFactory's and alias them at initialisation time.

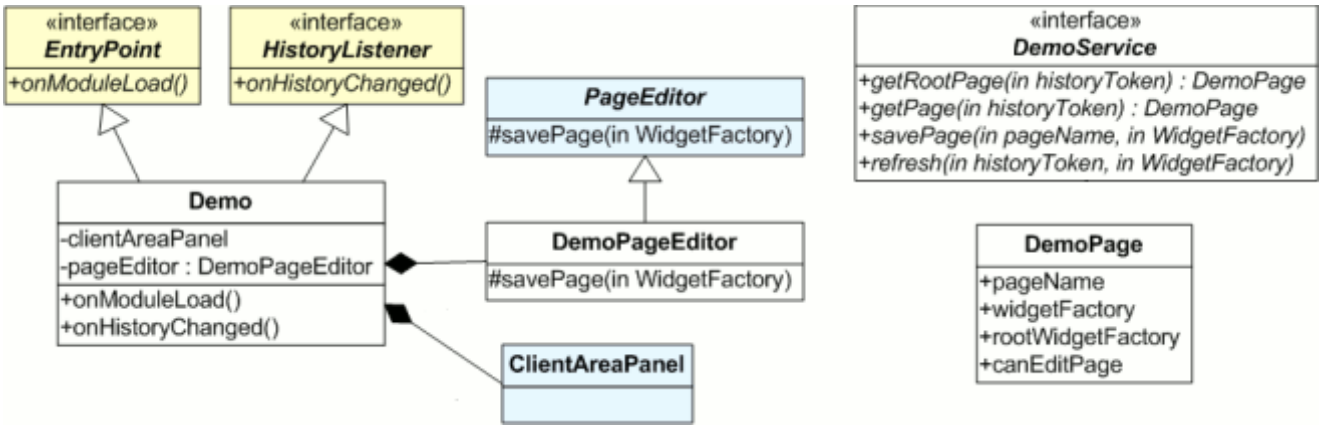
As mentioned in the Javadoc comment in the code fragment, fields annotated with DoNotPersist or DoNotSendToServer are not included in the XML. This is useful when fields in a PortletFactory are used only on the client side (DoNotSendToServer) or are used to pass information back to the server (DoNotPersist) that should not not end up in page XML files.

Chapter 4. Application Structure

This chapter describes how to pull the different parts of the framework together to create a complete application using one of the the bundled demos (demos/main in the distribution) as an example.

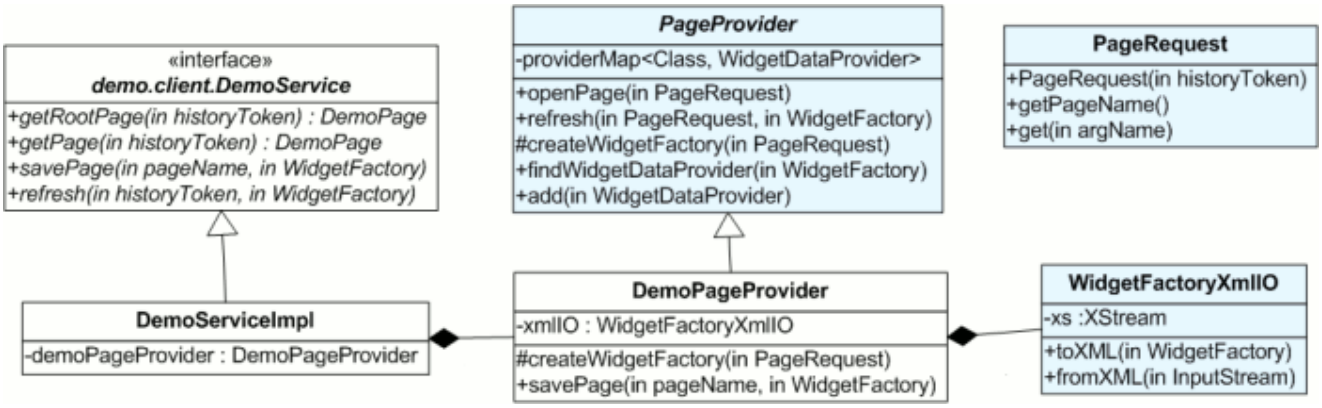
4.1. Classes

The main client side classes (main.client package) are shown in the class diagram below (blue classes are part of the GWT portlets framework, yellow classes are part of GWT):



- `Demo` The EntryPoint (main) class
- `DemoService` The service interface (Async version not shown)
- `DemoPage` DTO used to transfer WidgetFactory trees (pages) and related information from the server to client. A real application might include information on the currently logged on user etc.
- `DemoPageEditor` The page editor for in browser layout editing, knows how to save pages to the server
- `ClientAreaPanel` (framework) Container to fill the whole browser client area and layout child containers

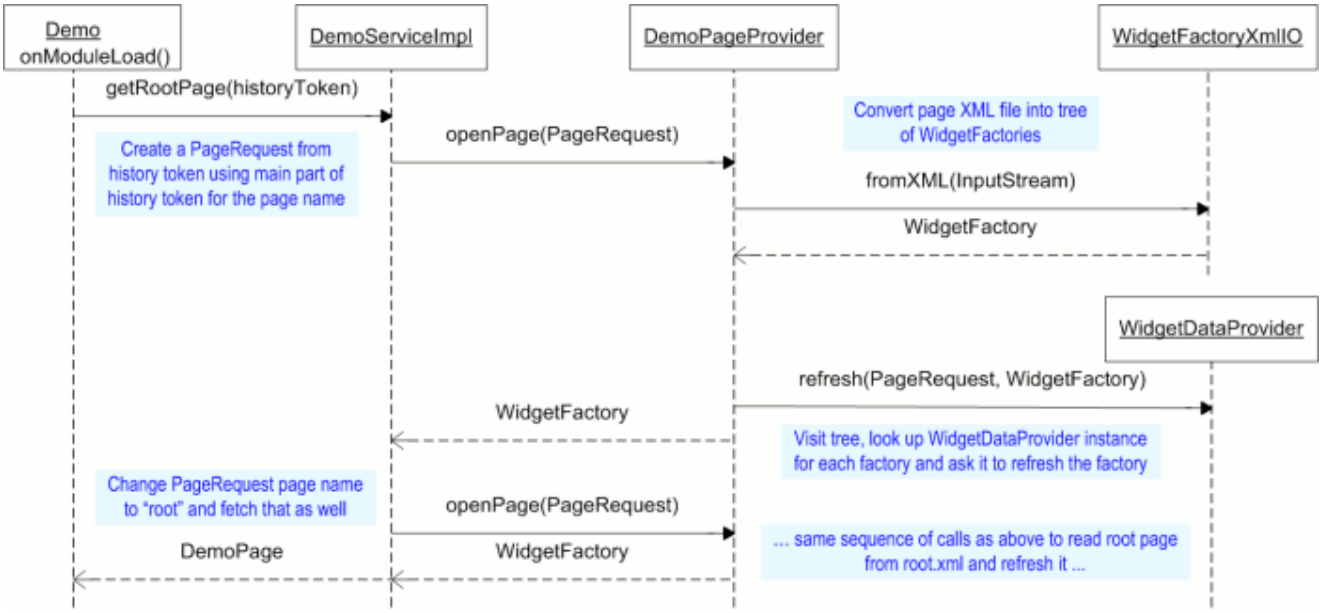
The server side classes (main.server package):



- `DemoServiceImpl` GWT RemoteServiceServlet implementing demo.client.DemoService methods
- `DemoPageProvider` Loads and saves "pages" (widget and portlet layouts) of the application from xml files and refreshes portlets
- `WidgetFactoryXmlIO` (framework) Used to convert WidgetFactory trees to/from XML using XStream

- `PageRequest` (framework) Analogous to an HttpServletRequest. It parses the history token into a page name and parameters and provides a store for per-request attributes (to share data between WidgetDataProviders) and the actual servlet request, response and context

The following (abridged) sequence diagram shows how the EntryPoint class for the demo application bootstraps the GUI from onModuleLoad():



The DemoPage DTO contains a WidgetFactory tree for the root page and for the page loaded for the history token. It a real application it would also contain information about the currently logged on user and so on. The rootWidgetFactory is used to create a tree of Widgets that is added to the ClientAreaPanel. The root page tree should contain a PagePortlet (and other widgets and portlets).

```
public class DemoPage implements Serializable {
    public String pageName;
    public WidgetFactory widgetFactory;
    public WidgetFactory rootWidgetFactory;
    public boolean canEditPage; // is page editable?
}
```

The PagePortlet portlet displays the current page by listening for PageChangeEvents and swapping out a new tree of widgets in a content area on page changes. If the page is editable then an edit button is displayed that launches the online layout editor on click. After the ClientAreaPanel has been populated with the root page, a PageChageEvent containing the widgetFactory tree (page) for the history token is broadcast to all the widgets in the tree. This is picked up by the PagePortlet.

```
<!-- Part of root.xml from the demo -->
<LayoutPanel styleName="" limitMaximize="false">
  ...
  <PagePortlet styleName="portlet-page"
        appTitle="GWT Portlets Demo"
        prefix="GWT Portlets Demo: "/>
  ...
```

Each time the history token changes (onHistoryChanged()), Demo calls DemoServiceImpl.getPage(historyToken) and the returned DemoPage DTO contains the page for the history token widgetFactory (rootWidgetFactory is null). A PageChangeEvent is broadcast and the PagePortlet displays the new page.

4.2. Event Broadcasting

The BroadcastManager singleton supports broadcasting of application wide events to widget trees. Only widgets that implement BroadcastListener receive broadcast events. This mechanism makes it easy for "global" events to be communicated to all parts of the GUI in a decoupled way.

```
package org.gwtportlets.portlet.client.event;
...
public class BroadcastManager {
...
    /**
     * Send the object to all widgets from the RootPanel down.
     */
    public void broadcast(Object ev) {...}

    /**
     * Send the object up to the logical parent of w (and its logical parent
     * and so on) until a widget with no logical parent is reached.
     */
    public void broadcastUp(Widget w, Object ev) {...}
...
```

For example the Demo application broadcasts a PageChangeEvent when the history token changes and a new page is loaded from the server. The PagePortlet uses this event to display the new page. It in turn broadcasts a PageTitleChangeEvent when it receives a WidgetChangeEvent from the widget on the page supplying the title. The PageTitlePortlet updates its title bar when it receives the event.

```
package org.gwtportlets.portlet.client.ui;
...
public class PagePortlet extends ContainerPortlet implements BroadcastListener {
...
    public void onBroadcast(Object ev) {
        if (ev instanceof PageChangeEvent) {
            onPageChange((PageChangeEvent)ev);
        } else if (ev instanceof WidgetChangeEvent
                && ((WidgetChangeEvent)ev).getSource() == titlePortlet) {
            updateTitle();
        }
    }
...
    private void updateTitle() {
        ...
        BroadcastManager.get().broadcast(new PageTitleChangeEvent(this, title));
    }
...
```

```
package org.gwtportlets.portlet.client.ui;
...
public class PageTitlePortlet extends Portlet implements BroadcastListener {
    private Label label = new Label("Page Title");
    ...
    public void onBroadcast(Object ev) {
        if (ev instanceof PageTitleChangeEvent) {
            label.setText(((PageTitleChangeEvent)ev).getPageTitle());
        }
    }
...
```

All Portlets must call boardcastUp with a WidgetChangeEvent when their title or flags change. This happens automatically when a Portlet refreshes itself. TitlePanel uses this event to show or hide a loading spinner, to update its title bar and to decide which buttons to display (configure, refresh etc.).

Applications can broadcast their own objects instead of having to couple components together directly (e.g. a LoggedOnUserPortlet might display a name and role from a LoggedOnEvent).

The BroadcastManager also supports non-widget listeners for broadcast events:

```
package org.gwtportlets.portlet.client.event;
...
public class BroadcastManager {
...
    /**
     * Add listener to be notified on calls to broadcast before the
     * event is dispatched to the widget tree.
     */
    public void addObjectBroadcastListener(BroadcastListener l) {...}
    public void removeObjectBroadcastListener(BroadcastListener l) {..}
...
```

4.3. The WidgetRefreshHook Singleton

The framework does not have its own service interface. Instead it relies a WidgetRefreshHook singleton to refresh a Portlet or Widget with new data from the server. This is normally set in onModuleLoad and just invokes a service method:

```
package main.client;
...
public class Demo implements EntryPoint {
    ...
    public void onModuleLoad() {
        WidgetRefreshHook.App.set(new WidgetRefreshHook() {
            public void refresh(Widget w, WidgetFactory wf,
                    AsyncCallback<WidgetFactory> cb) {
                DemoService.App.get().refresh(History.getToken(), wf, cb);
            }
            public void onRefreshCallFailure(Widget w, Throwable caught) {
                Window.alert("Refresh failed: " + caught);
            }
        });
        ...
```

The implementation of the service method delegates to the PageProvider to refresh the WidgetFactory tree. This simple mechanism makes it possible to refresh any portlet (actually any tree of Widgets implementing WidgetFactoryProvider) with new data without having to a new service method and code a specific async call.

```
package main.server;
...
public class DemoServiceImpl extends RemoteServiceServlet
        implements DemoService {
    private DemoPageProvider pageProvider;
    ...
    public WidgetFactory refresh(String historyToken, WidgetFactory wf) {
        pageProvider.refresh(createPageRequest(historyToken), wf);
        return wf;
    }

    private PageRequest createPageRequest(String historyToken) {
        PageRequest req = new PageRequest(historyToken);
        req.setServletConfig(getServletConfig());
        req.setServletRequest(getThreadLocalRequest());
        req.setServletResponse(getThreadLocalResponse());
        return req;
    }
...
```

4.4. WidgetFactory Trees and Data Providers

The Demo application service implementation refreshes WidgetFactory trees by delegating to the DemoPageProvider as show above. The code that handles the refresh is in the PageProvider base class and is the same code that runs after a page XML file has been read:

```
package org.gwtportlets.portlet.server;
...
public abstract class PageProvider {
...
    /** Refresh the data in the widget factory tree starting at top. */
    public void refresh(final PageRequest req, final WidgetFactory top) {
        top.accept(new WidgetFactoryVisitor() {
            public boolean visit(WidgetFactory wf) {
                WidgetDataProvider p = findWidgetDataProvider(wf);
                if (p != null) {
                    try {
                        p.refresh(wf, req);
                    } catch (Exception e) {
                        handleRefreshException(req, top, wf, e);
                    }
                }
                return true;
            }
        });
    }

    public WidgetDataProvider findWidgetDataProvider(WidgetFactory wf) {
        return providerMap.get(wf.getClass());
    }

    public void add(WidgetDataProvider p) {
        Class<? extends WidgetFactory> key = p.getWidgetFactoryClass();
        if (key == null) {
            throw new IllegalArgumentException("null not supported");
        }
        providerMap.put(key, p);
    }
...
```

Each WidgetFactory in the tree is visited and refreshed by a WidgetDataProvider. The provider is found by a simple map lookup using the class of the factory as the key. Note that WidgetDataProvider implementations need to be thread safe.

Chapter 5. Layouts

GWT Portlets provides an absolute positioning framework with pluggable layout managers (similar to Swing and other thick client GUI toolkits). This chapter explains how LayoutPanel and its default RowLayout work.

5.1. RowLayout

The default layout of a LayoutPanel is RowLayout. This flexible layout manager supports arranging widgets in a horizontal row (hence the name) or in a vertical column. In both cases the spacing between widgets in pixels is controlled by an int spacing property (default is 4 pixels). LayoutPanel uses the column mode by default.

The size of each widget is controlled by a RowLayout.Constraints instance with the following properties:

- `float size` The size of the widget in pixels (e.g. 100.0) or the fraction (0.3 for 30%) of available space it should take up
- `float weight` Weighting used to allocate extra space proportionally among widgets with weight > 0
- `int maxSize` Maximum size of the widget in pixels or 0 for no limit
- `String overflow` Value for overflow CSS style attribute to control scrollbars and clipping of the widgets content. Use one of the constants from LayoutConstraints: VISIBLE (don't clip), HIDDEN (clip), SCROLL (always show scrollbars) and AUTO (show scrollbars if needed but see warning below)

Once a widget has been positioned by a RowLayout the following additional read only properties are available:

- `int actualSize` The actual size of the widget in pixels
- `int extraSize` The extra space allocated to it in pixels (according to its weight)

In row mode the size sets the width of each widget and the height is the height of the container. In column mode the size sets the height of each widget and the width is the width of the container.

Widgets that implement HasMaximumSize are centered in the rectangle assigned to them if its width or height exceeds the maximums.

The interactive RowLayout demo provides an environment to experiment with RowLayout and its constraints.

5.2. Using LayoutPanel

Container (implemented by LayoutPanel) has several convenient add methods that make creating many popular layouts easier:

```
package org.gwtportlets.portlet.client.ui;
...
public  class LayoutPanel extends ComplexPanel implements Container {
...
    /** Add a widget with layout constraints. This does not redo the layout.  */
    public void add(Widget widget, LayoutConstraints constraints) {...}

    /** Add a widget with FloatLayoutConstraints. This does not redo the layout. */
    public void add(Widget widget, float constraints) {...}

    /** Add a widget with StringLayoutConstraints. This does not redo the layout. */
```

```
    public void add(Widget widget, String constraints) {...}
...
```

The first method adds the widget with the specified constraints. The second and third add the widget with float and string constraints. Layouts may use these constraints to create their own specific constraints (e.g. RowLayout.Constraints) in a reasonable way (typically by using the string or float as a constructor argument). In many cases it is possible to avoid creating RowLayout.Constraints instances.

The examples below are reproduced in the main demo.

This example creates a typical "buttons on top of scrolling body region" layout:

```
LayoutPanel p = new LayoutPanel(); // has RowLayout in column by default
p.add(buttons, 22); // new RowLayout.Contraints(22): size=22 weight=0 maxSize=0
                    // overflow=hidden
p.add(body); // new RowLayout.Constraints(): size=0 weight=1.0 maxSize=0 overflow=auto
p.layout();
```

This is example has a sidebar on the left using 20% of available space with auto scrollbars, a 20 pixel margin on the right without scrollbars and the rest of the space for the body with auto scrollbars:

```
LayoutPanel p = new LayoutPanel(false); // use RowLayout in row
p.add(sidebar, 0.2f); // new RowLayout.Contraints(0.2f): size=0.2 weight=0.0 maxSize=0
                      // overflow=auto
p.add(body); // new RowLayout.Constraints(): size=0 weight=1.0 maxSize=0 overflow=auto
p.add(margin, 20);  // new RowLayout.Contraints(20): size=20 weight=0.0 maxSize=0
                    // overflow=hidden
p.layout();
```

Here is "border layout":

```
LayoutPanel inner = new LayoutPanel(false); // row
inner.add(west, 0.2f);
inner.add(center);
inner.add(east, 0.2f);

LayoutPanel outer = new LayoutPanel(); // column
outer.add(north, 0.2f);
outer.add(inner, LayoutConstraints.HIDDEN); // avoid scrollbars in scrollbars
outer.add(south, 0.2f);
outer.layout();
```

These examples depend on the constructors for RowLayout.Constraints which are designed to make the common cases simple:

```
package org.gwtportlets.portlet.client.layout;
...
public class RowLayout implements Layout {
...
    public static class Constraints implements LayoutConstraints {
    ...
        /** Size=0.0, weight=1.0, MaxSize=0 */
        public Constraints(String overflow) {...}

        /** MaxSize=0, if size < 1.0 overflow=AUTO else overflow=HIDDEN */
        public Constraints(float size, float weight) {...}

        /** Weight=0.0, maxSize=0, if size < 1.0 overflow=AUTO else overflow=HIDDEN */
        public Constraints(float size) {...}

        /** Size 0.0, weight 1.0, maxSize=0, overflow=AUTO */
        public Constraints() {...}
```

```
...
```

5.3. Layout Tips

5.3.1. Scrollbars

Generally it is best to avoid placing regions with overflow=auto (i.e. scrollbars if needed) inside each other. In some older browsers scrollbars may appear when not needed when the outer region is made smaller.

5.3.2. Tables

Tables with padding around the outer TD elements end up taking up more space that what is assigned to them by the framework. This causes unnecessary scrollbars. Likewise a table with padding and width of 100% placed inside a DIV will get scrollbars. One solution is to avoid padding on TDs on the outer edges of the table. The FormBuilder class uses CSS styles on the first and last rows and columns in the table to achieve this effect.

This problem and the "narrow tables exploding" problem can also be solved by wrapping the table in a DIV (SimplePanel) as shown in this example:

```
LayoutPanel p = new LayoutPanel(); // widgets in a column
p.add(buttons, 22);
SimplePanel wrapper = new SimplePanel();
wrapper.add(table);
p.add(wrapper);
p.layout();
```

The width of wrapper is set to the width of the LayoutPanel and the table assumes its natural width inside it. Without the wrapper the table would be as wide as the LayoutPanel. This approach also avoids the problems with table TD padding and scrollbars.

5.3.3. Margins

The framework does not consider margins when laying out widgets. Use the layout spacing property and widget padding and borders to create space between widgets.

5.3.4. PositionAware interface

Widgets implementing PositionAware are notified by a call to boundsUpdated() when their position and/or size is changed. The Chart widget from the FreeMemoryPortlet uses this mechanism to size its Google Chart to fit the available space:

```
private class Chart extends Image implements PositionAware ... {
    public void boundsUpdated() {
        Rectangle r = LDOM.getContentBounds(this); // area inside our borders and padding
        // r.width, r.height == area available for chart
        ...
```

5.4. The LDOM Class

LayoutPanel and Layouts use static methods in LDOM to query and set position and size related properties for widgets and elements. The methods in this class take borders and padding into account when needed. There are different LDOMImpl implementations for different browsers.

The most important methods are shown in extract below:

```
package org.gwtportlets.portlet.client.layout;
...
public class LDOM {
...
    /**
     * Position the widget. If it implements {@link PositionAware} then it is notified
     * of this change. The width and height are adjusted to account for the
     * borders and padding of the widget if needed. Note that its margin is not
     * considered.
     */
    public static void setBounds(Widget w, int left, int top, int width, int height)
          {...}
    public static void setBounds(Widget w, Rectangle r) {...}

    /**
     * Get a bounding rectangle for w in browser client area coordinates.
     */
    public static Rectangle getBounds(Widget w) {...}

    /**
     * Get a bounding rectangle for the content area of w in browser client
     * area coordinates. This area excludes space used by borders and padding.
     */
    public static Rectangle getContentBounds(Widget w) {...}
...
```

Chapter 6. The Page Editor

The framework supports editing of pages (actually any tree of Containers and Portlets) in the browser. The application using the framework controls how the page editor is launched and what happens when the user "saves" a page or tree. The demo application uses the PagePortlet to display pages and this portlet displays a spanner icon on the bottom right hand corner of the client area for editable pages.

The appropriate code fragments from the demo are shown below:

```
package demo.client;
...
public class Demo implements EntryPoint ... {
...
    // The pageEditor is responsible for editing and saving pages (extends PageEditor)
    private DemoPageEditor pageEditor = new DemoPageEditor();
...
    private void onPageChange(final DemoPage p) {
        ...
        // The page change event knows how to edit the current page
        PageChangeEvent pce = new PageChangeEvent(this) {
            public void editPage(Container container) {
                ...
                pageEditor.startEditing(getPageName(), container);
            }
        };
        pce.setPageName(p.pageName);
        pce.setEditable(p.canEditPage);
        pce.setWidgetFactory(p.widgetFactory);

        // Send the event to every AppEventListener in the container tree.
        // The PagePortlet uses this event to change the widget tree in the
        // 'content area' of the application and to display the gear icon
        // for editable pages
        BroadcastManager.get().broadcast(pce);
    }
...
```

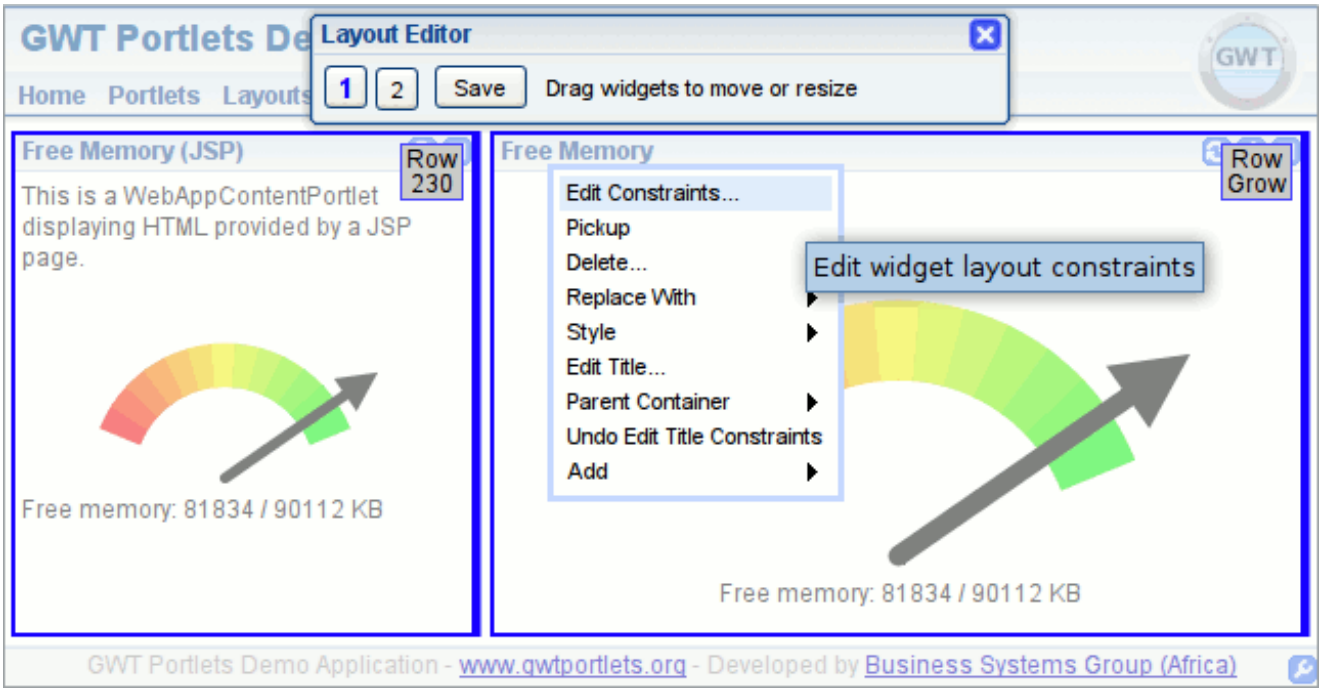
The PagePortlet calls editPage on the PageChangeEvent when the user clicks the gear icon for an editable page and the demo starts editing using its PageEditor subclass:

```
package demo.client;
...
public class DemoPageEditor extends PageEditor {
...
    protected void savePage(WidgetFactory wf, AsyncCallback callback) {
        DemoService.App.get().savePage(getPageName(), wf,
                new AsyncCallback() {
            public void onFailure(Throwable caught) {
                Window.alert("Oops " + caught);
            }
            public void onSuccess(Object result) {
                Window.alert("Saved");
            }
        });
    }
}
```

The only thing required from a PageEditor subclass is a savePage method. Override other methods if you need to customize the editor further.

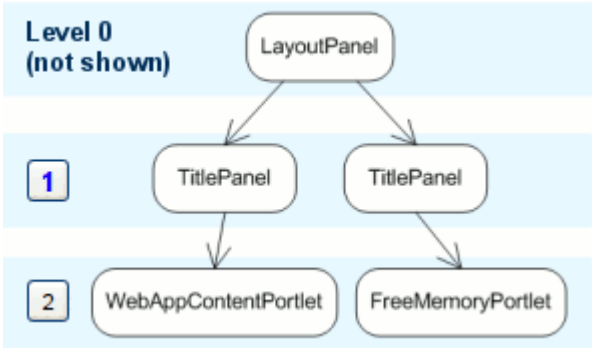
6.1. Using The Editor

The following screenshot (taken from a modifled version of the demo home page) shows the major components of the editor:



The Dialog titled "Layout Editor" displays a button for each level in the Container tree being edited ("1", "2" etc.) with the selected level highlighted.

The image below shows how levels in the editor correspond to levels in the container tree. You can click the buttons or use the mouse wheel to change levels. There is also a context sensitive message explaining what do ("Drag widgets to move or resize") and a save button. Closing this dialog stops editing.



Each widget on the current level on the tree is outlined by a blue rectangle. These rectangles can be dragged around to move widgets to different positions at the same level in the tree. The thick edge of the rectangle resizes the widget and left click opens a context sensitive menu (not all the options listed below are visible in the screenshot):

- *Edit Constraints*... Opens a dialog to edit layout constraints for the widget
- *Pickup* Pickup the widget and click to drop it somewhere else. Change levels (use the mouse wheel) while "holding" a widget to move widgets between levels in the tree
- *Delete*... Delete the widget

- *Replace With* > Replace the widget with a different widget. If the new widget is a Container then the widget is placed inside the container. This is very useful for "splitting" the space occupied by a widget into a row or column and for putting widgets inside TitlePanel's
- *Style* > Set the CSS style for the widget. The list of styles can be changed by overriding PageEditor.getStyleNamesFor(Widget)
- *Edit...* Edit settings for the widget that was clicked (in this case a TitlePanel)
- *Configure...* Configure the Portlet that was clicked (if it supports configure e.g. FreeMemoryPortlet)
- *Parent Container* > Edit settiings for the parent container of the clicked widget
- *Undo* Undo the last action
- *Redo* Redo the last undone action
- *Add* > Select a new widget or container and click to drop it

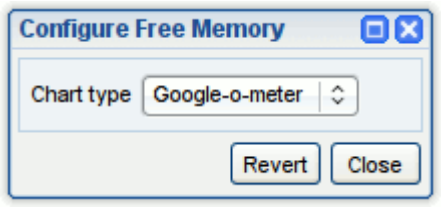
Don't forget to click *Save* to save changes.

Chapter 7. UI Widgets

The framework includes useful portlets and the widgets needed for its own UI (to avoid having to depend on other GWT libraries). The aim of the GWT Portlets framework is to make it easier to produce modular, decoupled business applications using GWT, not to create a widget library.

7.1. Dialog and CssButton

Dialog is a replacement for the standard GWT dialog box. It includes a title bar with maximize/restore and close buttons, content area, button bar, is styled using CSS and image sprites, is themable, prevents the application from receiving events (even mouse overs and so on) when modal, triggers close when escape is presssed and absolutely positions its contents.



CssButton is a Button subclass styled using a CSS background image sprite. It is lightweight (rendered using a single BUTTON element) and supports rollover. It selects different background sprites based on the width of the button avoiding scaling effects.

Here is a code fragment taken from FreeMemoryPortlet that creates its configuration dialog:

```
package demo.client.ui;
...
public class FreeMemoryPortlet extends Portlet {
...
    public void configure() {
        final ListBox type = ...
        FormBuilder b = new FormBuilder();
        b.label("Chart type").field(type).endRow();

        final Dialog dlg = new Dialog();
        dlg.setText("Configure " + getWidgetName());
        dlg.setWidget(b.getForm());
        dlg.addButton(new CssButton("Revert", new ClickHandler() {
            public void onClick(ClickEvent ev) {...}
        }, "Undo changes"));
        dlg.addCloseButton();
        dlg.showNextTo(this);
    }
...
```

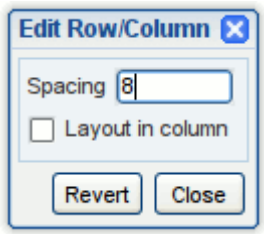
The content area of the dialog can be populated with a single widget by calling setWidget (like a standard GWT DialogBox) or multiple widgets can be added (getContent().add(Widget,...)). The setWidget method wraps widgets with a TABLE element in a SimplePanel (DIV) styled to add 4px padding.

The body of the Dialog is a RefreshPanel. This will display a AJAX loading pizza inside the dialog if it contains a Portlet and the portlet is refreshed.

The showNextTo method will position the dialog next to another widget. If there is more space to the right or left then the dialog will be positioned there, otherwise it is placed below or above. It will center the dialog if there is not enough space anywhere.

7.2. FormBuilder

FormBuilder is not actually a widget itself, it creates a form (labels and fields etc.) based on a GWT FlexTable



It keeps track of the current row and column in the table and creates labels and fields using standard styles. Some methods add new cells and others operate on the most recently added cell as shown in the following code fragment:

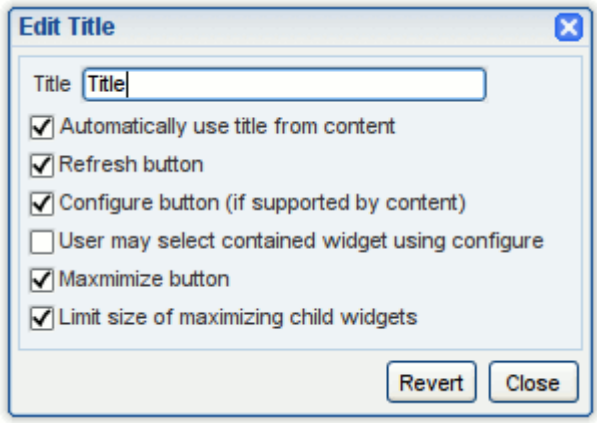
```
CheckBox column = new CheckBox("Layout in column");
TextBox spacing = new TextBox();

FormBuilder b = new FormBuilder();
b.label("Spacing").field(spacing).endRow();
b.field(column).colspan(2).endRow(); // checkbox spans 2 columns
FlexTable form = b.getForm();
```

FormBuilder styles the table so that the spacing between TDs inside is even (default is 4px) but the outer TDs (first row, last row, first column and last column) do not have any padding on the outside. This makes it easier to nest forms, to maintain consistent spacing and avoid problems with tables and spacing inside scrolling regions.

7.3. TitlePanel

TitlePanel is a Portlet and Container that contains other portlets. Each of the portlets in the home page of the demo is contained in a TitlePanel. It provides a title bar, refresh, configure and maximize buttons and displays an AJAX loading pizza when the first Portlet it contains is refreshing.



The dialog on the right is displayed by clicking a TitlePanel in the page editor and selecting Edit Title... from the popup menu. The options are as follows:

- *Automatically use title from content* Display the title provide by the first portlet in the panel or use the title captured here
- *Refresh button* Display a refresh button in the title bar that refreshes the contents of the panel
- *Configure button* Display a configure button if the first portlet in the panel supports configure. This invokes the portlets configure() method

- *Maxmimize button* Display a maximize button that will expand the TitlePanel up to the next maximize limit point (typically whole page content area)
- *Limit size of maximizing child widgets* Contained widgets with maximize support (e.g. nested TitlePanel's) will maximize to the boundarys of this TitlePanel before getting bigger
- *User may select contained widget using configure* Users may select a different Portlet to go into this TitlePanel using the configure button. If the currently contained Portlet supports configure then the user is prompted to replace it with something else or configure it

7.4. WebAppContentPortlet

The WebAppContentPortlet displays any content served from the web application including JSP pages, static HTML, servlets and so on. It is configured with the path to the content as shown in the main demo (demos/main).

7.5. MenuPortlet

The MenuPortlet displays a lightweight HTML menu generated from an HTML template served from the web application (a JSP page, static HTML etc.). It is configured with the path to the template as shown in the main demo (demos/main)

The template for the demo application (menu.html) is shown below:

```
<a href="#home">Home</a>
<a href="#portlets()">Portlets</a>
<a href="#commands()">Commands</a>

<div id="portlets">
    <a href="#hello_world">Hello World</a>
    <a href="#free_memory">Free Memory</a>
    <a href="#web_app_content">Web App Content</a>
    <a href="#row_layout">Row Layout</a>
</div>

<div id="commands">
    <a href="#command_demo">Command Demo</a>
    <a href="#command1(arg1,arg2)">Command1</a>
    <a href="#command2()">Command2</a>
    <a href="#command3()">Command3</a>
</div>
```

- Menu items are normal links and can contain history tokens ('`#home`'), external links and commands ('`#command1(arg1,arg2)`'). Note that other HTML elements may be present
- Submenus are identified by top level DIVs with id attributes e.g. `<div id="portlets">`
- Menu items that activate sub menus have a matching href attribute and round brackets e.g. `<a href="#portlets()">Portlets</a>`
- History token links with round brackets that do not activate sub menus broadcast a CommandEvent to all widgets when clicked

The CommandDemoPortlet on command demo page displays the most recent CommandEvent received.

7.6. PagePortlet

PagePortlet listens for PageChangeEvent's and displays the widgets for the new page in its "content" area. If the page is editable then an edit button is displayed that launches the page editor on click.

It updates the browser window title and broadcasts a PageTitleChangeEvent when the title of the first Portlet on the page changes. The application title and prefix used to construct the browser window title can be configured by selecting *Configure...* for on the PagePortlet in the PageEditor.

It is styled `portlet-page` by default.

7.7. PageTitlePortlet

PageTitlePortlet listens for PageTitleChangeEvent's and updates a title label. It is styled `portlet-page-title` by default.

7.8. ToolButton

ToolButton displays a small icon defined by a CSS background image sprite with rollover and disabled support.

```
ToolButton edit = new ToolButton(ToolButton.CONFIGURE, "Edit Page", new ClickListener() {
    public void onClick(Widget sender) {...}
});
```

7.9. ShadowPanel

ShadowPanel adds a fuzzy shadow to a single widget. The look of the shadow is controlled by the theme.

```
Widget w = ...
ShadowPanel sp = new ShadowPanel(w);
```

Chapter 8. Theme Support

The framework themes its widgets using a singleton Theme instance, theme CSS files and Javascript maps. Several themes are bundled with the framework. The CSS for BlueGradient is included in gwt-portlets.css as it is the default theme. The selected theme stored in a cookie.

8.1. The Theme Singleton

The Theme class supports querying available themes and changing the theme:

```
package org.gwtportlets.portlet.client.ui;
...
public class Theme {
    /** Get the singleton Theme instance. */
    public static Theme get() {...}

    /** Get the name of the currently selected theme. */
    public String getCurrentTheme() {...}

    /** Get the names of the available themes.  */
    public String[] getThemes() {...}

    /**
     * Change to a different theme or NOP if the theme is already active.
     * Expects to find a gwt-portlets-<name>.css stylesheet (unless the default
     * 'BlueGradient' theme is selected) and an optional
     * gwt_portlets_<name> Javascript object with overrides for dimensions
     * of dialog headers and whatnot. <b>NB: This method reloads the
     * application if the theme is changed.</b>
     */
    public void changeTheme(String name) {...}
...
```

The demo includes a ThemeListPortlet to display and change the theme.

The list of available themes may be changed by defining a Javascript object in your bootstrap HTML file:

```
<script type="text/javascript">
var gwt_portlets = {
    themes: "BlueGradient, LightBlue, MyTheme"
};
</script>
```

8.2. Creating Themes

Defining a new theme that has the same dimensions for UI elements (e.g. dialog title bar height) involves creating just a CSS file. Here is `gwt-portlets-LightBlue.css`:

```
.portlet-dialog-header-bg,
.portlet-dialog-footer-bg {
    background-image: url( "img/portlet-dialog-blue.png" );
}

.portlet-dialog-sides-bg {
    background-image: url( "img/portlet-dialog-sides-blue.png" );
}

.portlet-title-header-bg {
    background-image: url( "img/portlet-caption-blue.gif" );
}
```

```
.portlet-dialog-content {
    background-color: #dae7f6;
}

.portlet-dialog-content-body {
    border: 1px solid #99bbe8;
}

.portlet-dialog-buttonbar {
    background-color: #dae7f6;
}

.portlet-title-body {
    border-left: 1px solid #99bbe8;
    border-right: 1px solid #99bbe8;
    border-bottom: 1px solid #99bbe8;
}
```

If your theme changes the dimensions of things (e.g. dialog title bar height) then you also need to define a Javascript object (typically in your bootstrap HTML file):

```
<script type="text/javascript">
var gwt_portlets_MyTheme = {
    titleBarHeight: 20;
    titleBarLeftWidth : 6;
    ...
};
</script>
```

The framework widgets ask the Theme singleton to apply the selected theme, typically when their style is set:

```
public class Dialog extends PopupPanel {
...
    public void setStyleName(String style) {
        ...
        Theme.get().updateDialog(this);
        ...
```

```
public class Theme {
...
    /** Configure the dimensions of the dialog. */
    public void updateDialog(Dialog dlg) {
        EdgeRow header = dlg.getHeader();
        header.setDimensions(0, dialogHeaderHeight, dialogHeaderLeftWidth,
            dialogHeaderRightWidth);
        ...
```

If you want your own widgets to use the same mechanism you may need to extend Theme and call Theme.set(Theme instance) in onModuleLoad.

Chapter 9. Integrating with Spring

Spring is a platform for building Enterprise Java applications. It provides lifecycle management and wiring for application components (among other features). A Spring / GWT Portlets application will typically do the following:

- Annotate WidgetDataProviders with `@Service` or `@Repository` to make them Spring beans
- Make the application PageProvider a Spring bean with an `@Autowired` WidgetDataProvider[] property to discover all WidgetDataProviders

A complete demo application using Spring and JPA (Java Persistence API) will added to the distribution soon.