



Web 2.0 -- Advanced AJAX Applications with the Google Web Toolkit

Building rich interfaces for Web applications

Authors: High Performance On Demand Solutions
(HiPODS) Latin America Team

Web Address: ibm.com/websphere/developer/zones/hvws

Management contact: Sergio Loza loza@br.ibm.com

Sales contact: Marcio Barbosa marcio@br.ibm.com

Technical contact: Ari Dias arid@br.ibm.com

Date: February 16, 2007

Status: Version 1.0

Abstract: This paper is a guide to using the Google Web Toolkit (GWT) to integrate rich user interface components with remote services. It includes examples of creating a paged table with editable fields and a personalized dialog box. The management of events and service calls for data access, as well as peculiar aspects of the API and tips about its use, are discussed. GWT is an excellent option to improve productivity when developing AJAX applications for the Web.

Contents

Contents.....	2
Introduction	3
Setting up the development platform	3
Building the application components.....	4
Working with components	8
Google Web Toolkit services	12
Putting it all together	16
Running the application	17
Conclusion.....	18
Appendix: Panels and widgets	18
References	19

Introduction

Web 2.0 is the term coined for the growing Internet trend characterized by online collaboration and sharing among users. Web 2.0 is changing the way people interact with Web applications, leading nine out of ten Web sites to review their implementation of the request/wait/response paradigm. Technologies like asynchronous JavaScript and XML (AJAX) enhance the user experience of accessing the Web, meeting standards that Web application developers have sought for many years.

The success of AJAX occurred when advanced Web applications such as Google Maps and Gmail became generally available. Google offers a framework -- the Google Web Toolkit (GWT) -- to facilitate the development of advanced AJAX applications.

This paper tells how to build AJAX applications, including the use of personalized components that extend the application program interface (API) of GWT. To demonstrate the benefits of this technology, we include examples of creating a paged table with editable fields and a personalized dialog box. We discuss the management of events and service calls for data access, as well as peculiar aspects of the API and tips about its use.

Setting up the development platform

If you want to perform the steps in this paper:

1. Download IBM® Rational® Application Developer for WebSphere® Software Version 6.0 for your integrated development environment (IDE).
2. Download IBM® Cloudscape™ for your database.
3. Download the source code for the sample. The sample is an Eclipse project with everything set up and ready to be imported into Rational Application Developer. To make tests easier, the download file includes a class that creates the database and populates it with random data.
4. Execute the class `util.DatabaseBuilder.java` as a Java™ program.
5. Download GWT from www.code.google.com/webtoolkit and update the project libraries configuration.

If you want to create a personalized development environment to build a new project, set the GWT home directory in the path and run these commands in List 1 from a new folder:

List 1. Two commands to create GWT Eclipse shell project and a sample application

```
projectCreator -eclipse GWTComponents
applicationCreator -eclipse GWTComponents com.ibm.hipods.gwt.client.MyApplication
```

`projectCreator` generates a shell Eclipse project
`applicationCreator` generates the Hello World application

Import the project you create into Rational Application Developer. The created project contains a runtime configuration to execute the application.

Building the application components

The sample application is a contact list composed of a paged table and a photo area; see Figure 1. The application can create a dialog box with information on a contact selected from the table. Note some peculiar aspects included in the application:

- The table has editable fields, sensitive to mouse clicks and the Enter key.
- Data persistence is implemented in a transparent way. When the editable field does not have the focus any longer, data is updated in the database without any user interference.
- The dialog box can be closed with the use of the mouse or the keyboard (Enter or Esc).

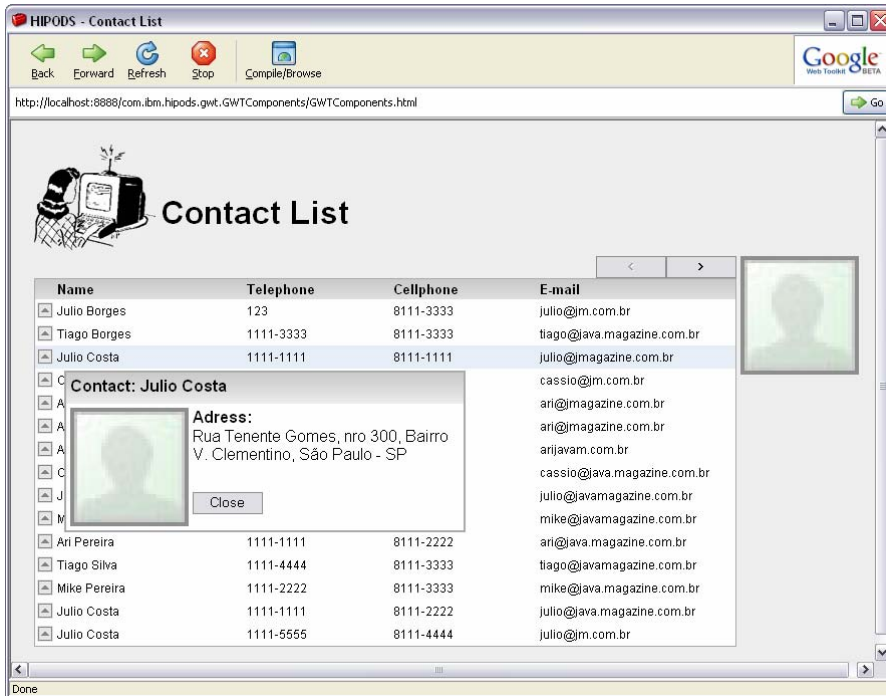


Figure 1. Contact list

Figure 2 shows the three main packages that comprise the project:

`com.ibm.hipods.gwt.client` contains Java code that is converted in JavaScript and executed in the browser

`com.ibm.hipods.gwt.server` contains the service classes

`com.ibm.hipods.gwt.public` contains images, CSS files, HTML pages, and the like

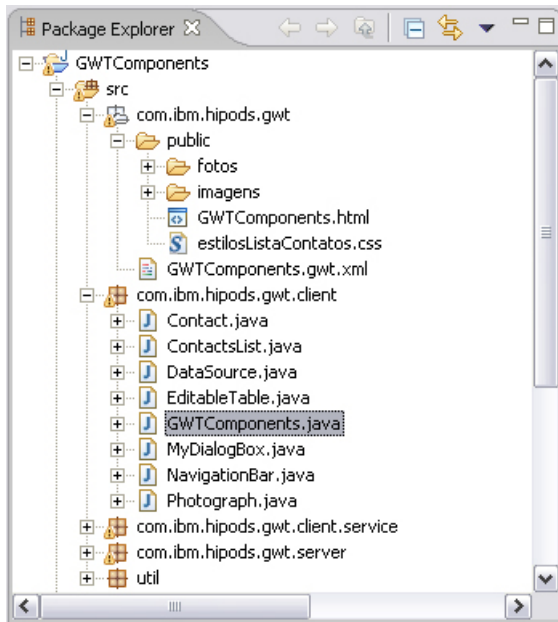


Figure 2. Project structure

Additional project files are:

`com.ibm.hipods.gwt.public.GWTComponents.html` is an HTML file with tags indicating where the AJAX components are inserted and the location where the GWT JavaScript libraries (`gwt.js`) are imported from. This file has a `<meta />` element defining the module that must be loaded. GWT information calls this file *host page*.

`com.ibm.hipods.gwt.client.GWTComponents.java` is the class responsible for starting the components and placing them in the HTML file. GWT calls this class *entry point*.

`com.ibm.hipods.gwt.GWTComponents.gwt.xml` defines the GWT module and service settings.

List 2 shows the content of `GWTComponents.html`. This file includes two components referred by their IDs (contacts and photograph) inside an HTML table and the name of the module that will be loaded.

List 2. GWTComponents.html

```
<html>
<head>
  <title>HIPODS - Contact List</title>
  <meta name='gwt:module' content='com.ibm.hipods.gwt.GWTComponents'>
  <link rel="stylesheet" href="styles.css" type="text/css">
</head>
<body>
  <script language="javascript" src="gwt.js"></script>
  <iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
<table>
  <tr>
    <td></td>
    <td valign="bottom">
```

```

        <h1>Contact List</h1>
        </td>
</table>
<table>
  <tr>
    <!-- contact list -->
    <td id="contacts"></td>
    <!-- contact photograph -->
    <td id="photograph" valign="top"></td>
  </tr>
</table>
</body>
</html>

```

More details about the other two files are discussed in future sections.

Photograph component

The first component created shows the photo of the selected contact and extends the class `Image` of GWT. The `Photograph` component is used twice in the application: beside the contact table and inside the dialog box that pops out when a contact is selected. An extra function of the `Photograph` class (List 3) is to display a standard photo if the contact does not have an associated photo.

List 3. Photograph.java

```

public class Photograph extends Image implements LoadListener {

    public Photograph(Contact contact) {

        //set the photo url
        setUrl("fotos/" + Long.toString(contact.getContactId()) + ".jpg");

        //set the photo size
        setWidth("110");
        setHeight("110");

        //set css style
        setStyleName("photograph");

        //set this instance as the load listener
        addLoadListener(this);
    }

    public void onError(Widget sender) {
        setUrl("fotos/default_photo.jpg");
    }

    public void onLoad(Widget sender) {}
}

```

Note these aspects of List 3:

- The `Photograph` constructor receives an object of the contact class (List 4). It includes definitions of the size, CSS style, and Web address of the image.
- The methods `onError()` and `onLoad()` are used in the implementation of the `LoadListener` interface. `onError()` is accessed when there is a problem loading the image,

for example, when the Web address of the image is wrong. `onLoad()` is accessed when the image is loaded correctly and is empty because there is no need for any treatment.

- The class instance is defined as its own listener in the constructor with `addLoadListener(this)`.

List 4. Contact.java

```
/**
 * Implements IsSerializable to warn GWT this class
 * will be sent between client and server.
 */
public class Contact implements IsSerializable {

    public long contactId;
    public String name;
    public String telephone;
    public String cellphone;
    public String email;
    public String address;

    public Contact() {}

    // getters and setters
}
```

MyDialogBox component

The `DialogBox` component is the part of the GWT that allows pop-up windows with information and/or warnings. In this sample application, an extended class of `DialogBox` is created to use characteristics such as title area, dragging possibility, and its modal characteristic that blocks the rest of the application while the window is active.

The dialog box layout is shown in Figure 1. CSS was used to provide title and border formatting. The style definition is in file `gwt.css`, and imported through page `GWTComponents.html`. Most of the GWT graphic components (widgets) have a predefined CSS style, which means that the component uses it as long as you define it in the CSS file. The `DialogBox` uses two styles, one for the window and another for the title area: `gwt-DialogBox` and `gwt-DialogBox.Caption`.

List 5 shows the component code of `MyDialogBox`. The title, size, and location of the dialog box were defined in the constructor. The photo, content, and button were placed inside a `DockPanel` component. To understand more about panels and other GWT graph components, see *Appendix: Panels and widgets*.

List 5. MyDialogBox.java

```
public class MyDialogBox extends DialogBox {

    private final static int LEFT_MARGIN = 80;
    private final static int TOP_MARGIN = (Window.getClientHeight() - 256) / 2;
    private final static String WINDOW_SIZE = "400px";

    public MyDialogBox(Image image, String title, String content) {
        setText(title);

        //DialogBox position
        setPopupPosition(LEFT_MARGIN, TOP_MARGIN);
        setWidth(WINDOW_SIZE);
    }
}
```

```

//build the close button and its listener
Button btnClose = new Button("Close", new ClickListener() {
    public void onClick(Widget sender) {
        hide();
    }
});

//create the base panel (DockPanel) and define the DialogBos layout
DockPanel panelBase = new DockPanel();
panelBase.add(image, DockPanel.WEST);
panelBase.add(btnClose, DockPanel.SOUTH);
panelBase.add(new HTML(content), DockPanel.CENTER);

//space between the components
panelBase.setSpacing(4);

//set the DialogBox base panel
setWidget(panelBase);
}

public boolean onKeyDownPreview(char key, int modifiers) {
    switch (key) {
        case KeyboardListener.KEY_ENTER:
        case KeyboardListener.KEY_ESCAPE:
            hide();
            break;
    }
    return true;
}
}
}

```

Also note that a `ClickListener` was instantiated in the creation of the closing button. Its function is to close the dialog box through the `hide()` method taken from the super class when there is a click event (`onClick()`). At last, the `onKeyDownPreview()` method, which deals with the keyboard events, defines that the dialog box is closed while pressing Enter or Esc, also by the `hide()` method.

Working with components

GWT `Composite` class organizes the build of more complex components. It works like a component *wrapper* hiding the public methods of the *packed* component. The `Composite` is usually set with a panel (for example, `DockPanel`), which has several components. This is the way they can be treated as a single element. Three components based on the `Composite` class are created:

`NavigationBar` is composed by a `HorizontalPanel` and contains two buttons and one label.

`EditableTable` is composed by a `DockPanel` with the navigation bar on top and a centered *grid*.

`ContactList` creates an `EditableTable` component with a data source and set the column names.

Note that a composite may have other composite objects. This case is found in `ContactList` and `EditableTable`. Additional details on how to create the three components described above are discussed next.

NavigationBar component

The navigation bar has three elements: two buttons (forward and back) and one Label (see List 6). Its constructor sets the elements positioning inside a `HorizontalPanel` and receives a data source. More information about the `DataSource` object is provided in *Google Web Toolkit services*.

List 6. NavigationBar.java

```
public class NavigationBar extends Composite implements ClickListener {

    private HorizontalPanel horizontalPanel = new HorizontalPanel();

    //the second parameter is the click listener
    public Button nextBtn = new Button(">", this);
    public Button previewBtn = new Button("<", this);
    public Label statusMsg = new Label();
    private DataSource dataSource;
    private int actualPage = 0;

    public NavigationBar(DataSource dataSource) {

        this.dataSource = dataSource;

        horizontalPanel.setHorizontalAlignment(HasHorizontalAlignment.ALIGN_LEFT);
        horizontalPanel.setWidth("100%");
        horizontalPanel.add(statusMsg);
        horizontalPanel.setCellWidth(statusMsg, "100%");
        horizontalPanel.add(previewBtn);
        horizontalPanel.add(nextBtn);

        previewBtn.setEnabled(false);
        initWidget(horizontalPanel);
    }

    public void onClick(Widget sender) {
        if (sender == nextBtn) {
            actualPage++;
        }
        else if (sender == previewBtn) {
            actualPage--;
        }
        setStatus("Loading ...");
        dataSource.setPage(actualPage);
    }

    public void setStatus(String status) {
        this.statusMsg.setText(status);
    }

    public void enable(boolean lastPage) {
        nextBtn.setEnabled(lastPage);
        previewBtn.setEnabled(actualPage > 0);
        setStatus("");
    }
}
```

The main functions of the navigation bar are to manage clicks on the back (<) and forward (>) buttons and to show the message *Loading...* while the table is filled up. The bar is shown above the contact list (see Figure 1).

The `NavigationBar` component implements `ClickListener` and registers itself as a listener in the moment of its buttons instantiation (passing a reference of itself as a second parameter) in order to capture the clicks on the forward and back buttons.

As soon as one of the buttons is clicked, the method `onClick()` is accessed. The component updates the attribute `actualPage`, changes the label message to `Loading...` and accesses the method `setPage()` from the `DataSource`. Note that the navigation bar does not refer to the `EditableTable` component. It interacts only with `DataSource`, which is responsible for updating the objects connected to it.

EditableTable component

The `EditableTable` component is the most complex component of the application. The most important part of its code is shown in List 7. The internal structure of the component is built inside a `DockPanel` with a `NavigationBar` component in the north position and a `Grid` (from API of GWT) in the center position where the contact data is displayed.

The constructor instantiates the navigation bar with a data source, keeps the number of entries on each page, sets the instance as a `Grid` listener (to capture the clicks on it), and determines the layout of the components inside the `DockPanel`.

The method `onCellClicked()` is accessed when any click event happens inside the grid. If the click happens in the first column, a dialog box with the contact address is displayed. A click in any other column makes the correspondent cell editable.

The `showDialogBox()` method instantiates the dialog box with the contact photo and address already configured. The dialog box is shown through the method `show()`.

`editCell()` substitutes the `Label` on the clicked cell by a `TextBox` (from the API of GWT) for edition. This `TextBox` has two listeners: one for focus events, `FocusListener` and another for keyboard events, `KeyboardListener`. When the `TextBox` loses its focus or if the user presses `Enter`, the method `updateRecord()` is accessed to save the changes in the database.

At last, the method `updateRecord()` substitutes the `TextBox` for a `Label`, updates the correct field of the `Contact` object, and saves the changes through the `DataSource`. In addition, it keeps an internal mapping between the grid lines and shown contacts to identify which entry was selected. That is how the `Contact` object is selected when its correspondent line is selected as well.

List 7. EditableTable.java

```
public class EditableTable extends Composite implements TableListener {
    /* attributes: panelBase, grid, contacts, navigationBar, totalPerPage,
     * selectedLine, photograph, dataSource.
     */

    public EditableTable(DataSource dataSource, String[] columnNames,
        int totalPerPage)
    {
        // ... attributes configuration
        // ... datasource configuration

        /* set this as the table listener */
        grid.addTableListener(this);
    }
}
```

```

//layout and positioning configuration
panelBase.add(navigationBar, DockPanel.NORTH);
panelBase.add(grid, DockPanel.CENTER);
initWidget(panelBase);

}

/**
 * Receives the cell click events
 */
public void onCellClicked(SourcesTableEvents sender, int row, int cell) {

    if (row == 0) return;
    selectLine(row);
    showPhotograph(row);
    if (cell == 0) {
        showDialogBox(row);
        return;
    }

    if (cell > 1) {
        Widget widget = grid.getWidget(row, cell);
        if (widget instanceof TextBox) { return; }
        editCell(row, cell, widget);
    }
}

private void showDialogBox(int row) {
    Contact contact = getContact(row);
    Photograph fotografia = new Photograph(contact);

    //build the custom dialog
    MyDialogBox dialogBox = new MyDialogBox(fotografia, "Contact: "
        + contact.getName(), "<b>Adress: </b> <br />" + contact.getAddress());

    dialogBox.show();
}

private void editCell(final int row, final int cell, Widget labelCelula) {
    final TextBox textBox = new TextBox();
    textBox.setText(((Label) labelCelula).getText());
    grid.setWidget(row, cell, textBox);

    //add keyboard event treatment
    textBox.addKeyboardListener(new KeyboardListener() {
        public void onKeyDown(Widget sender, char keyCode, int modifiers) {}

        public void onKeyPress(Widget sender, char keyCode, int modifiers) {}

        public void onKeyUp(Widget sender, char keyCode, int modifiers) {
            switch (keyCode) {
                case KeyboardListener.KEY_ENTER:
                    updateRecord(row, cell, textBox.getText());
                    break;
            }
        }
    });

    //when it loses the focus it updates the database with a new value
    textBox.addFocusListener(new FocusListener() {...});
}

private void updateRecord(int row, int column, String cellContent) {

```

```

//substitute the TextBox for a Label
Label label = new Label();
TextBox textBox = (TextBox) grid.getWidget(row, column);
label.setText(textBox.getText());
grid.setWidget(row, column, label);

//get the selected
Contact contact = getContact(row);
switch (column) {
    case 2:
        contact.setTelephone(cellContent);
        break;
    case 3:
        contact.setCellphone(cellContent);
        break;
    case 4:
        contact.setEmail(cellContent);
        break;
}
//update the database
dataSource.updateContact(contact);
}

private void showPhotograph(int row){...}
public void updateData(List contacts){...}
private void setGridLayout(){...}
private Contact getContact(){...}
private void selectLine(int row){...}
}

```

ContactList component

The ContactList component gathers a data source (object DataSource) and an EditableTable component (see List 8). The data source gets the number of entries on a single page. The editable table gets the names of the columns, the number of entries on a single page, and the DataSource itself.

List 8. ContactList.java

```

public class ContactList extends Composite {

    private EditableTable editableTable;
    private DataSource dataSource;

    public ContactList(int totalPerPage) {
        String[] nomesColunas = new String[] { "", "Name", "Telephone", "Cellphone",
            "E-mail" };
        this.dataSource = new DataSource(totalPerPage);
        this.editableTable = new EditableTable(dataSource, nomesColunas,
            totalPerPage);
        initWidget(editableTable);
    }
}

```

Google Web Toolkit services

The application uses the DataSource class to access the contact data in the database. This class uses the GWT remote services infrastructure. This section introduces these services before explaining the code.

Figure 3 shows how an AJAX call works and illustrates the steps when an XMLHttpRequest call is made.

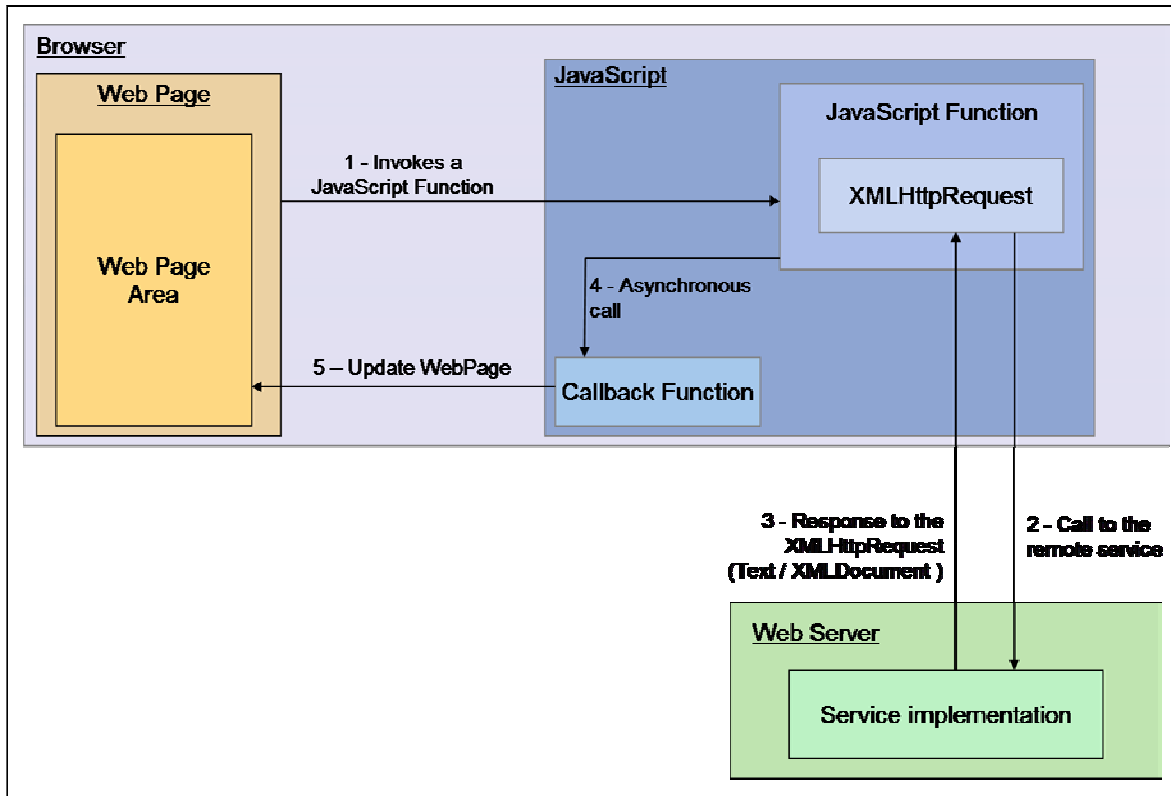


Figure 3. An AJAX call

GWT makes these steps transparent for the developer because it includes an infrastructure to work on remote procedure calls (RPCs). It facilitates the Java object trade between the client (the browser) and the server. All calls are asynchronous, so the graphic interface is free for eventual changes while the data does not come thus optimizing application performance and helping parallelism, even without using multiple threads. Part of the code is executed in the server and part is executed in the client to handle graphic interface events.

A *service interface* has to be defined to create a remote service at GWT and this interface should define all the method signatures. The method implementation is in charge of a class that is executed in the server as a *Servlet*. This class implements the service interface and extends the class *RemoteServiceServlet* from the API of GWT.

Another interface may also be created: the *asynchronous interface*. Its methods are the same as the service interface, but it has an extra parameter: a callback object that receives the service response as soon as it is processed. Another difference is that all its methods return `void`.

Created services

The service includes the interface `ContactServices`, the asynchronous interface `ContactServicesAsync`, and the implementation class `ContactServicesImpl`.

The service interface has two methods: `getContacts()` to search for the contact list and `updateContact()` to update the contact edited (see List 9).

List 9. ContactServices.java

```
public interface ContactServices extends RemoteService {
    //service methods
    public List getContacts(int initialPosition, int totalPerPage) throws
    Exception;

    public void updateContact(Contact contact) throws Exception;

    public static final String ENTRY_POINT = "/contactServices";

    public static class Util {...}
}
```

When this interface is changed, the asynchronous interface ContactServicesAsync must be updated by creating similar methods to the ContactServices ones (see List 10).

List 10. ContactServicesAsync.java

```
public interface ContactServicesAsync {
    //service methods
    public void getContacts(int initialPosition, int totalPerPage, AsyncCallback
    callback);

    public void updateContact(Contact contact, AsyncCallback callback);
}
```

The class ContactServicesImpl has the implementation code for the methods defined in the interface ContactServices (see List 11). Note that a direct access to a database occurs. For a business application, we recommend using a connection pool and some design pattern as DAO (data access object) to encapsulate all access to the data source.

List 11. ContactServicesImpl.java

```
public class ContactServicesImpl extends RemoteServiceServlet implements
    ContactServices
{
    private static final long serialVersionUID = 1L;

    public ContactServicesImpl() {}

    public List getContacts(int initialPosition, int totalPerPage)
    throws Exception
    {
        Connection conn = null;
        Statement stmt = null;
        try {
            List temp = new ArrayList();

            String sql = "select contact_id, name, telephone, cell_phone, email,
            address from contact";
            conn = getConnection();
            stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
            ResultSet rs = stmt.executeQuery(sql);
            rs.beforeFirst();
            rs.afterLast();
            rs.absolute(initialPosition + 1);
            for (int i = 0; i < totalPerPage && rs.next(); i++) {
                temp.add(buildContact(rs));
            }
        }
    }
}
```

```

        return temp;
    }
    catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
    finally {
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    }
}

public void updateContact(Contact contact) throws Exception {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        String sql = "update contact set name = ?, telephone = ?, cell_phone = ?,
email = ? where contact_id = ?";
        conn = getConnection();
        stmt = conn.prepareStatement(sql);
        stmt.setString(1, contact.getName());
        stmt.setString(2, contact.getTelephone());
        stmt.setString(3, contact.getCellphone());
        stmt.setString(4, contact.getEmail());
        stmt.setLong(5, contact.getContactId());
        stmt.executeUpdate();
    }
    catch (Throwable e) {
        e.printStackTrace();
    }
    finally {
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    }
}

public Connection getConnection() throws Exception {...}
private Contact buildContact(ResultSet rs) throws Exception {...}
}

```

The last step adds service descriptions into the module setup file, `com.ibm.hipods.gwt.GWTComponents.gwt.xml`. This file is like the one in List 12.

List 12. GWTComponents.gwt.xml

```

<module>
    <!-- Inherit the core Web Toolkit stuff.          -->
    <inherits name='com.google.gwt.user.User' />

    <!-- Specify the app entry point class.          -->
    <entry-point class='com.ibm.hipods.gwt.client.GWTComponents' />

    <!-- Data access service -->
    <servlet path='/contactServices'
        class='com.ibm.hipods.gwt.server.ContactServicesImpl' />
</module>

```

Putting it all together

With all the services and components ready, it is time to put them together. Figure 4 shows how the components interact with the `DataSource`.

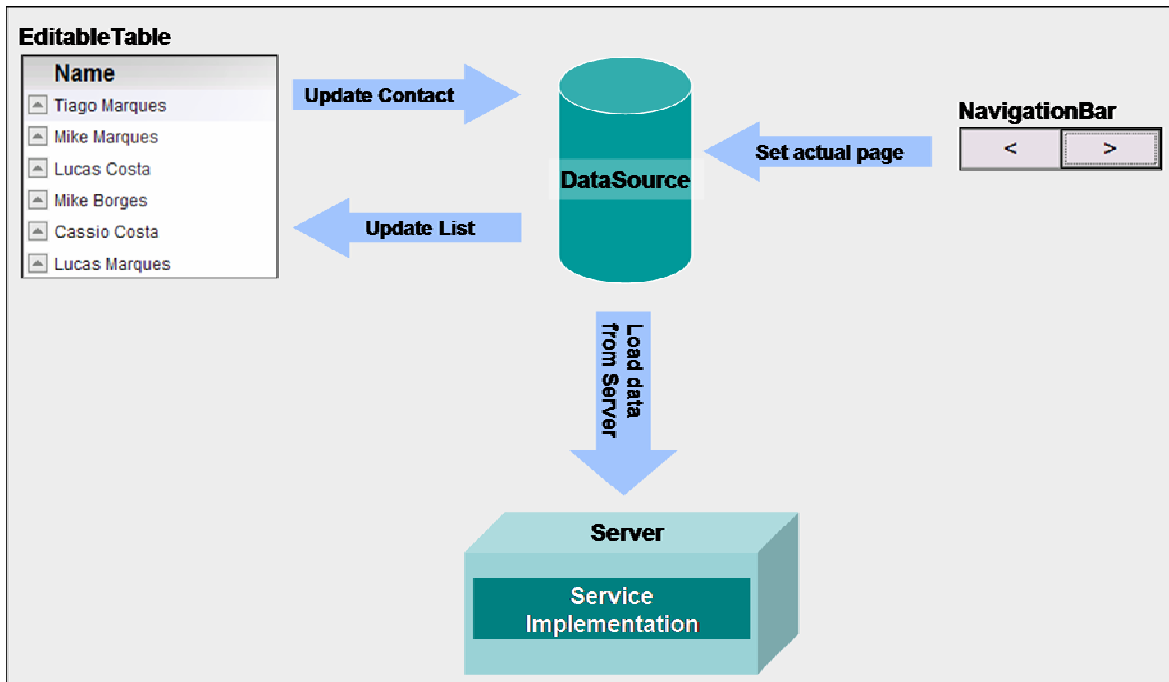


Figure 4. How components interact with the `DataSource`

Investigate data access by studying the data source code provided in List 13. The class `DataSource` is very simple. It contains two inner classes, `ContactServicesCallBack` and `UpdateCallBack`. Both have two methods: `onFailure()` for error treatment in the service call and `onSuccess()` to receive the response.

List 13. `DataSource.java`

```
public class DataSource {  
    private int totalPerPage;  
    private EditableTable editableTable;  
  
    public DataSource(int totalPerPage) {  
        this.totalPerPage = totalPerPage;  
        setPage(0);  
    }  
    public void setPage(int pageNumber) {  
        ContactServicesAsync service = ContactServices.Util.getInstance();  
        service.getContacts(pageNumber * totalPerPage, totalPerPage,  
            new ContactServicesCallBack());  
    }  
    public void setEditableTable(EditableTable editableTable) {  
        this.editableTable = editableTable;  
    }  
    public void updateContact(Contact contact) {  
        ContactServicesAsync service = ContactServices.Util.getInstance();  
        service.updateContact(contact, new UpdateCallBack());  
    }  
}
```

```

}

private class ContactServicesCallBack implements AsyncCallback {
    public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
    }
    public void onSuccess(Object result) {
        editableTable.updateData((List)result);
    }
}

private class UpdateCallBack implements AsyncCallback {
    public void onFailure(Throwable caught) {
        Window.alert("Error on update database.");
    }
    public void onSuccess(Object result) {}
}
}
}

```

The class `ContactServicesCallBack` receives the contact search result and sets up the table that holds the returned data. `UpdateCallBack` does not have any specific implementation because the service is an update without a server return. Errors are treated in a simple way in both: a pop-up window with a message appears. The methods `updateContact()` and `setPage()` make the service calls.

Let's return to the class `GWTComponents`. As shown in List 14, inside the method `onModuleLoad()` a `RootPanel` is searched with the id `contacts`. Remember that this id has been defined in the `GWTComponents.html` file and is where the contact list is placed. Then a `ContactList` component is added to the `RootPanel`. Doing so, the GWT starts the module and loads the contact list as soon as the HTML file is displayed.

List 14. `GWTComponents.java`

```

public class GWTComponents implements EntryPoint {
    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
        //list the contacts - 15 per page
        RootPanel.get("contacts").add(new ContactList(15));
    }
}
}

```

Running the application

To run this sample application:

1. Open the execution set-up menu (Run|Run).
2. In the configuration tree, select **Java Application>GWTComponents**
3. Click **Run**.
4. Confirm that the application is similar to Figure 1.

Conclusion

Google Web Toolkit is an excellent option to improve productivity when developing AJAX applications. It is aligned to Web 2.0 technologies and makes it easy to integrate Web applications that are AJAX based.

This paper introduced how to extend the GWT API and create new components. Some Web sites specialize in GWT components and they can help developers save a lot of time while coding new applications.

Appendix: Panels and widgets

The appendix briefly introduces the GWT API panels and widgets used by our application.

Panels

A *panel* is a container of *widgets*, which are the user interface components used to create layouts and define the positioning of graphic elements. Our application used three panels:

`HorizontalPanel`, `DockPanel`, and `RootPanel`.

`HorizontalPanel` organizes its components horizontally from left to right in a line. It can be compared to the `FlowLayout` from the Abstract Window Toolkit (AWT) API. Our application used this panel to organize the navigation bar buttons and message label.

`DockPanel` sets the position of its elements following a numeric rule as shown in Figure 5 (extract from the GWT information). As soon as it sets an element in the center position, no other element should be added to the panel.

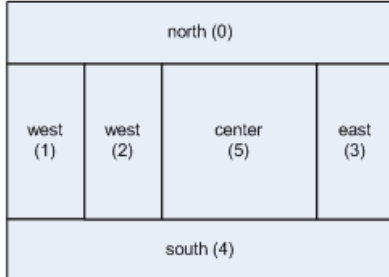


Figure 5. DockPanel positioning order

If no position is specified when an element is added, the last component added is positioned in the center. Our application used `DockPanel` as the base layout to the `MyDialogBox` and `EditableTable` components.

`RootPanel` is the last panel a component should be added to. Its location is defined in the HTML page (host page), and normally it is used an ID for identification. The `RootPanel` is instantiated by GWT and its instance can be obtained by two methods:

`RootPanel.get()` returns the default `RootPanel` for the application

`RootPanel.get(String id)` returns the `RootPanel` associated to an ID in the host page
Its most important function is to hold principal components.

Widgets

The GWT API has a collection of graphic components: text field, checkboxes, tables, menus, pop-up windows, navigation trees, grid, and others. Some details about them are presented below.

`TextBox` is a simple field for text input. Its default CSS style is `gwt-TextBox`. Main methods:

- `getText()` returns its content
- `getSelectedText()` returns selected text inside the text field
- `selectAll()` selects all the text inside the text field
- `setText(String)`: set the text field content
- `addChangeListener()`, `addClickListener()`, `addKeyboardListener()` add listener for content change and mouse and keyboard events

`Button` is a normal HTML button. Its default CSS is `gwt-Button`. Main methods:

- `click()` is similar to a mouse click
- `addClickListener()`, `addKeyboardListener()` add listeners for mouse and keyboard events

`Image` shows an image from a specific Web address. It has a default CSS style: `gwt-Image{}`.

Main methods:

- `setUrl()` sets the image Web address
- `addChangeListener()`, `addClickListener()`, `addKeyboardListener()` add listener for content change and mouse and keyboard events.

`Grid` is a resizable table that can contain text, HTML or widgets. It does not have a CSS style.

Main methods:

- `resize(int, int)` resizes the table;
- `setWidget(int linha, int coluna, Widget w)` add a widget in a specific cell
- `setText(int linha, int coluna, String str)` add a string in a specific cell
- `setHTML(int linha, int coluna, String str)` add HTML code in a specific table
- `getWidget()`, `getText()`, `getHTML()` return the widget, the string, or the HTML code from a specific cell
- `addTableListener()` adds a click listener.

`DialogBox` is a dialog box that can be dragged by the user. It has two CSS style: `gwt-DialogBox` and `gwt-DialogBox`.

`Caption` is one widget that we customized for our application. See the section *MyDialogBox component* for more details. Main methods:

- `setText()` sets a string as the content of the dialog
- `setHTML()` sets HTML code as the content of the dialog
- `addWidget()` set a widget as the content of the dialog

References

Google Web Toolkit, accessed at 12/19/2006 at <http://code.google.com/webtoolkit/>.

GWT Powered, accessed at 19/12/2006 at <http://gwtpowered.org>

Notices

Trademarks

The following are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

Cloudscape
IBM
Rational

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

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

Special notice

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environment do so at their own risk.

Any performance data contained in this document were determined in various controlled laboratory environments and are for reference purposes only. Customers should not adapt these performance numbers to their own environments as system performance standards. The results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.