Smart GWT™ Quick Start Guide

Smart GWT v6.1

Smart GWT™ Quick Start Guide

Copyright ©2014 and beyond Isomorphic Software, Inc. All rights reserved. The information and technical data contained herein are licensed only pursuant to a license agreement that contains use, duplication, disclosure and other restrictions; accordingly, it is "Unpublished-rights reserved under the copyright laws of the United States" for purposes of the FARs.

Isomorphic Software, Inc.
1 Sansome Street, Suite 3500
San Francisco, CA 94104
U.S.A.

Web: www.isomorphic.com

Notice of Proprietary Rights
The software and documentation are copyrighted by and proprietary to Isomorphic Software, Inc. ("Isomorphic"). Isomorphic retains title and ownership of all copies of the software and documentation. Except as expressly licensed by Isomorphic in writing, you may not use, copy, disseminate, distribute, modify, reverse engineer, unobfuscate, sell, lease, sublicense, rent, give, lend, or in any way transfer, by any means or in any medium, the software or this documentation.

1. These documents may be used for informational purposes only.
2. Any copy of this document or portion thereof must include the copyright notice.
3. Commercial reproduction of any kind is prohibited without the express written consent of Isomorphic.
4. No part of this publication may be stored in a database or retrieval system without prior written consent of Isomorphic.

Trademarks and Service Marks
Isomorphic Software, Smart GWT, SmartClient and all Isomorphic-based trademarks and logos that appear herein are trademarks or registered trademarks of Isomorphic Software, Inc. All other product or company names that appear herein may be claimed as trademarks or registered trademarks of their respective owners.

Disclaimer of Warranties
THE INFORMATION CONTAINED HEREIN IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT AND ONLY TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Contents

Contents i

How to use this guide v

Why Smart GWT? vii

More than Just Widgets – A Complete Architecture vii
Eliminates Cross-Browser Testing and Debugging vii
Complete Solution viii
Open, Flexible Architecture viii

1. Overview 1

Architecture 1
Capabilities and Editions of Smart GWT 2

2. Installation 3

Starting a New Project 3
Adding Smart GWT to an Existing Project 3
Server Configuration (optional) 4

3. Resources 5

Showcase 5
Java Doc 6
Developer Console 6
Community Wiki 11
FAQ 11

4. Visual Components 12

Component Documentation & Examples 12
Drawing, Hiding, and Showing Components 13
Size and Overflow 13
Handling Events 14

5. Data Binding 17

Databound Components 17
Fields 18
DataSources 22
Customized Data Binding 25
DataSource Operations 26
DataBound Component Operations 28
Data Binding Summary 29

6. Layout 30

Component Layout 30
Container Components 32
Form Layout 33

7. Data Integration 35

DataSource Requests 35
Smart GWT Server Framework 36
DSRequests and DSResponses 37
Request and Response Transformation 37
Criteria, Paging, Sorting and Caching 39
Authentication and Authorization 40
Relogin 41

8. Smart GWT Server Framework 42

DataSource Generation 42
Server Request Flow 45
Direct Method Invocation 47
DMI Parameters 48
Adding DMI Business Logic 48
Returning Data 52
Queuing & Transactions 53
Queuing, RESTHandler, and SOAs 55
Operation Bindings 55
Declarative Security 57
Declarative Security Setup 59
Non-Crud Operations 60
Dynamic Expressions (Velocity) 62
Server Scripting 65
Including Values from Other DataSources 67
SQL Templating 68
SQL Templating — Adding Fields 71
Why focus on `.ds.xml` files? 73
Custom DataSources 74

9. Extending Smart GWT 76

New Components 76
New Form Controls 78
Switching Theme 79
Customizing Themes 80

10. Tips 83

Beginner Tips 83
Architecture Tips 83
HTML and CSS Tips 85

11. Evaluating Smart GWT 87

Which Edition to Evaluate 87
Evaluating Performance 88
Evaluating Interactive Performance 90
Evaluating Editions and Pricing 91
A note on supporting Open Source 92

Contacts 93

How to use this guide

The *Smart GWT Quick Start Guide* is designed to introduce you to the Smart GWT™ web presentation layer. Our goals are:

- To have you working with Smart GWT components and services in a matter of minutes.

- To provide a conceptual framework, with pointers to more detail, so you can explore Smart GWT in your areas of interest.

This guide is structured as a series of brief chapters, each presenting a set of concepts and hands-on information that you will need to build Smart GWT-enabled web applications. These chapters are intended to be read in sequence—earlier chapters provide the foundation concepts and configuration for later chapters.

This is an *interactive* manual. You will receive the most benefit from this guide if you are working in parallel with the Smart GWT SDK—following the documented steps, creating and modifying the code examples, and finding your own paths to explore. You may want to print this manual for easier reference, especially if you are working on a single-display system.

We assume that you are somewhat acquainted with basic concepts of *web applications* (browsers, pages, tags), *object-oriented programming* (classes, instances, inheritance), and *user interface development* (components, layout, events). However, you do not need deep expertise in any specific technology, language, or system. If you know how to navigate a file system, create and edit text files, and open URLs in a web browser, you can start building rich web applications with Smart GWT today.

If you can't wait to get started, you can skip directly to *Installation* (Chapter 2) to create a Smart GWT project and begin *Resources* (Chapter 3) and *Visual Components* (Chapter 4). But if you can spare a few minutes, we recommend reading the introductory chapters first, for the bigger picture of Smart GWT goals and architecture.

Thank you for choosing Smart GWT, and welcome.

Why Smart GWT?

Smart GWT helps you to build and maintain more usable, portable, efficient web applications faster, propelled by an open, extensible stack of industry-tested components and services.

In this chapter we explore the unique traits of the Smart GWT platform that set it apart from other technologies with similar purpose.

More than Just Widgets – A Complete Architecture

Smart GWT provides an **end-to-end application architecture**, from UI components to server-side transaction handling.

The examples included with Smart GWT demonstrate the simplicity that can only be achieved by a framework that addresses both server- and client-side architectural concerns to deliver globally optimal solutions.

Smart GWT's UI components are carefully designed to maximize responsiveness and minimize server load, and Smart GWT's server components are designed around the requirements of high-productivity user interfaces.

Even if you adopt only part of the Smart GWT solution, you benefit from an architecture that takes into account the entire problem you need to solve, not just a part of it. Every integration point in the Smart GWT platform has been designed with a clear understanding of the requirements you need to fulfill, **and**, the solutions built into Smart GWT provide a "blueprint" for one way of meeting those requirements.

Eliminates Cross-Browser Testing and Debugging

Smart GWT provides a **clean, clear, object-oriented approach** to UI development that shields you from browser bugs and quirks.

Even if you need to create a totally unique look and feel, Smart GWT's simplified skinning and branding requires only basic knowledge of page styling, and you never have to deal with browser layout inconsistencies.

In contrast, lower-level frameworks that provide a thin wrapper over browser APIs can't protect you from the worst and most destructive of browser issues, such as timing-dependent bugs and memory leaks.

Smart GWT's powerful, component-oriented APIs give Smart GWT the flexibility to use whatever approach works best in each browser, so you don't have to worry about it.

This allows Smart GWT to make a simple guarantee: if there is a cross-browser issue, **it's our problem, not yours**.

Complete Solution

Smart GWT offers a **complete presentation layer** for enterprise applications: everything you need for the creation of full-featured, robust, high-productivity business applications.

The alternative—throwing together partial solutions from multiple sources—creates a mish-mash of different programming paradigms, inconsistent look and feel, and bizarre interoperability issues that no single vendor can solve for you.

Whether you are a software vendor or enterprise IT organization, it never makes sense to build and maintain a UI framework of your own, much less to own "glue code" tying several frameworks together. A single, comprehensive presentation framework gives you a competitive advantage by enabling you to focus on your area of expertise.

Open, Flexible Architecture

Because Smart GWT is built entirely with ***standard*** web technologies, it integrates perfectly with your existing web content, applications, portals, and portlets. You can build a state-of-the-art application from scratch, or you can upgrade existing web applications and portals at your own pace, by weaving selected Smart GWT components and services into your HTML pages.

By giving you both options, Smart GWT allows you to address a broader range of projects with a single set of skills. You can even reuse existing content and portlets by embedding them in Smart GWT user interface components. Smart GWT allows a smooth *evolution* of your existing web applications—you aren't forced to start over.

Next, Smart GWT is fully ***open*** to integration with other technologies. On the client, you can seamlessly integrate Java applets, Flash/Flex modules,

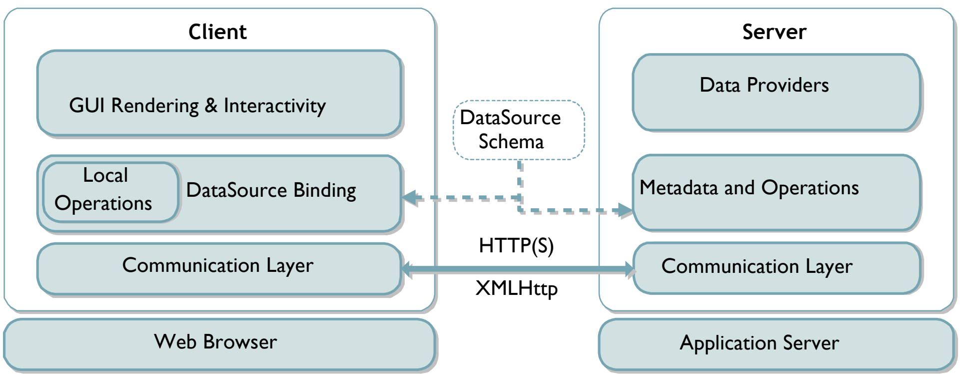
ActiveX controls and other client technologies for 3D visualization, desktop integration, and other specialized functionality. On the server, Smart GWT provides flexible, generic interfaces to integrate with any data or service tier that you can access through Java code.

Finally, Smart GWT is completely ***extensible***, all the way down to the web standards on which the system is constructed. If you can't do something "out of the box," you can build or buy components that seamlessly extend Smart GWT in any manner you desire.

1. Overview

Architecture

The Smart GWT architecture spans client and server, enabling Rich Internet Applications (RIAs) that communicate transparently with your data and service tiers.



Within the web browser, Smart GWT provides a deep stack of services and components for rich HTML5 / Ajax applications. For those using a Java-based server, Smart GWT provides a server-side framework that can be added to any existing Java web application.

The client- and server-based components have a shared concept of *DataSources*, which describe the business objects in your application. By working from a single, shared definition of the data model, client- and server-side components can coordinate closely to deliver much more sophisticated functionality "out of the box" than either a standalone client-based or server-based solution can deliver.

For example, validation rules are declared within the DataSource—these rules are then enforced client-side by Smart GWT Ajax components, and server-side by Smart GWT server components. Similarly, the set of valid operations on an object is declared in a DataSource, and this single declaration controls client-side behaviors like whether an editing interface is enabled, and controls security checks on the server for safe enforcement of business rules.

Using a DataSource as a shared data definition also greatly reduces redundancy between your user interface code and server-side code, increasing agility and reducing maintenance effort.

DataSources can be derived on-the-fly or as a batch process from other, pre-existing sources of metadata, such as annotated Java Beans and XML Schema, further reducing system-wide redundancy.

This concept of a DataSource as a shared client-side data definition can be used with or without the optional Smart GWT Java server components. However, if you do not use the Smart GWT server components, all server-side functionality of DataSources must be implemented and maintained by your team.

Finally, note that Smart GWT does not require that you adopt this entire architecture. You may choose to integrate with only the layers and components that are appropriate for your existing systems and applications.

Capabilities and Editions of Smart GWT

Smart GWT comes in several editions, and the features included in each of the editions are described on the SmartClient.com website at

http://www.SmartClient.com/product

The portions of this document that make use of Smart GWT server components require the Pro license or above. Certain features demonstrated or referred to in the document require Power or Enterprise Editions of SmartClient - this is mentioned in each case.

If you have downloaded the LGPL version, we recommend downloading the commercial trial version for use during evaluation. Prototypes built on the commercial edition can be converted to the LGPL version without wasted effort, but the reverse is not true—using the LGPL version for evaluation requires you to expend effort implementing functionality that is already part of the commercial version. For more details, see Chapter 11, *Evaluating Smart GWT*.

2. Installation

Starting a New Project

To get started quickly, use the “built-in-ds” sample project included in the Smart GWT SDK under `samples/built-in-ds`. Within this directory, see the `README.txt` file for instructions to import the project into the Eclipse IDE or to run the project from the command line using Apache *ant*.

Several other sample projects are provided that demonstrate integration with popular Java technologies. However, because it is the simplest and represents Smart GWT best practices, we recommend starting with the “built-in-ds” project unless you have a specific need to do otherwise. See Chapter 11, *Evaluating Smart GWT*, for details.

Do not start by importing the Showcase project. The Showcase is designed to host several hundred short-lived mini-applications and to demonstrate a variety of data binding approaches. It is not structured like a normal application and does not represent best practices for normal applications.

Adding Smart GWT to an Existing Project

If you wish to install Smart GWT into a pre-existing project, see the step-by-step setup guide (*http://www.smartclient.com/smartgwtee/javadoc/com/smartgwt/client/docs/SgwtEESetup.html*).

For purposes of this Quick Start, we strongly recommend using one of the sample projects, which have a pre-configured, embedded server and database.

Server Configuration (optional)

- You do not need to perform any server configuration for this Quick Start. However, if you choose to complete exercises in later chapters by connecting to an existing database, follow these additional steps: The *Smart GWT Admin Console* provides a graphical interface to configure database connections, create database tables from DataSource descriptors, and import test data. **Note:** Requires Smart GWT Server framework.

  To use the *Admin Console,* in your `gwt.xml` file, include the following imports:

  ```
  <inherits name="com.smartgwtee.SmartGwtEE"/>
  <inherits name="com.smartgwtee.tools.Tools"/>
  ```

  After adding these imports you then call `com.smartgwtee.tools.client.SCEE.openDataSourceConsole()`.

  ```
  IButton adminButton = new IButton("Admin Console");
  adminButton.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
          com.smartgwtee.tools.client.SCEE.openDataSourceConsole();
      }
  });
  adminButton.draw();
  ```

  Note: If you are using Pro, the method to call is `com.smartgwtpro.tools.client.SCPro.openDataSourceConsole()`. Make the corresponding adjustment for Power edition as well.

- Other server settings are exposed for direct configuration in the samples in the `server.properties` file. The `server.properties` file should be in your Eclipse CLASSPATH. Typically you do this by copying the file to the top of your `src` folder, so that Eclipse deploys it into `war/WEB-INF/classes`.

Settings that apply to servlets can be applied via the standard `web.xml` file. See the JavaDoc for each servlet for details.

If you have any problems installing or starting Smart GWT, try the Smart GWT Developer Forums at *forums.smartclient.com*.

3. Resources

Showcase

To start exploring, visit the Smart GWT EE Showcase at:

http://www.smartclient.com/smartgwtee/showcase/

The Showcase is your best starting point for exploring Smart GWT capabilities and code. For each example in the Showcase, you can view the source code by clicking on the View Source button in the upper right side of the example pane.

A second Showcase, focusing on capabilities common to both the LGPL and Pro/EE editions, exists at:

http://www.smartclient.com/smartgwt/showcase/

For all samples related to data binding, data loading and data integration, focus on the samples in the EE Showcase while you are learning. The data integration samples in the LGPL Showcase focus on concerns that become irrelevant if you adopt Smart GWT Pro or better.

To replicate the functionality of a Showcase sample in your own project, copy and paste code from the View Source window. **Do not** copy code from the samples/showcase folder in the Smart GWT SDK, as this code is designed specifically for running inside the specialized Showcase environment. The View Source window shows source code designed for standalone use.

Java Doc

The core documentation for Smart GWT is the *Smart GWT JavaDoc*. You can access the *Smart GWT JavaDoc* in any of the following ways:

Online:

smartclient.com/smartgwtee/javadoc/ (*client reference*)
smartclient.com/smartgwtee/server/javadoc/ (*server reference*)

In the SDK:

`docs/javadoc` (*client reference*)
`docs/server` (*server reference*)

There are two special packages in the client reference**:**

- **com.smartgwt.client.docs** contains conceptual overviews covering cross-cutting concerns that apply to multiple Smart GWT classes and packages. These appear in JavaDoc as Java Interfaces in order to allow interlinking with normal JavaDoc reference. Do not import this package—it is informational only.

- **com.smartgwt.client.docs.serverds** is reference for all properties that may be specified in `.ds.xml` file (see the *Data Binding* chapter). Do not import this package—it is informational only.

Developer Console

The Smart GWT Developer Console is a suite of development tools implemented in Smart GWT itself. The Console runs in its own browser window, parallel to your running application, so it is always available in every browser and in every deployment environment. Features of the Developer Console include:

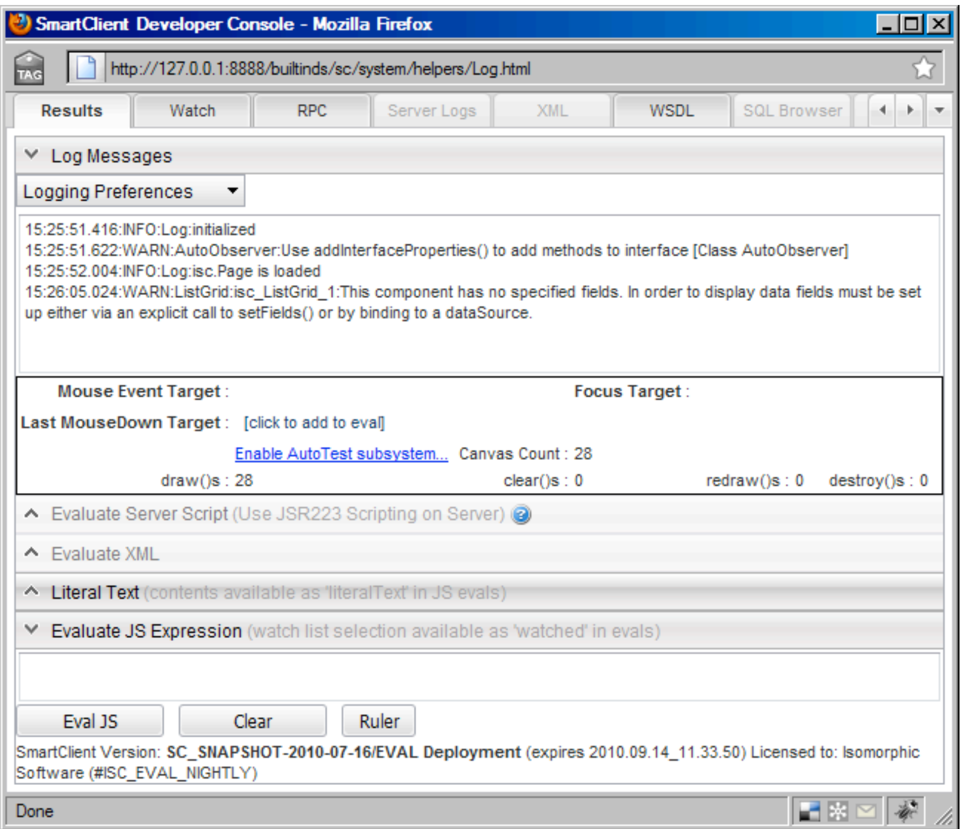
- logging & runtime diagnostics
- runtime component inspection
- tracing and profiling

To use the Developer Console, complete the following steps:

1. In your `*.gwt.xml` file, inherit the module `com.smartgwt.tools.SmartGwtTools`.

2. Open the Developer Console by typing the URL `javascript:isc.showConsole()` into the URL bar of the hosted mode browser or any standard browser.

The following window will appear:



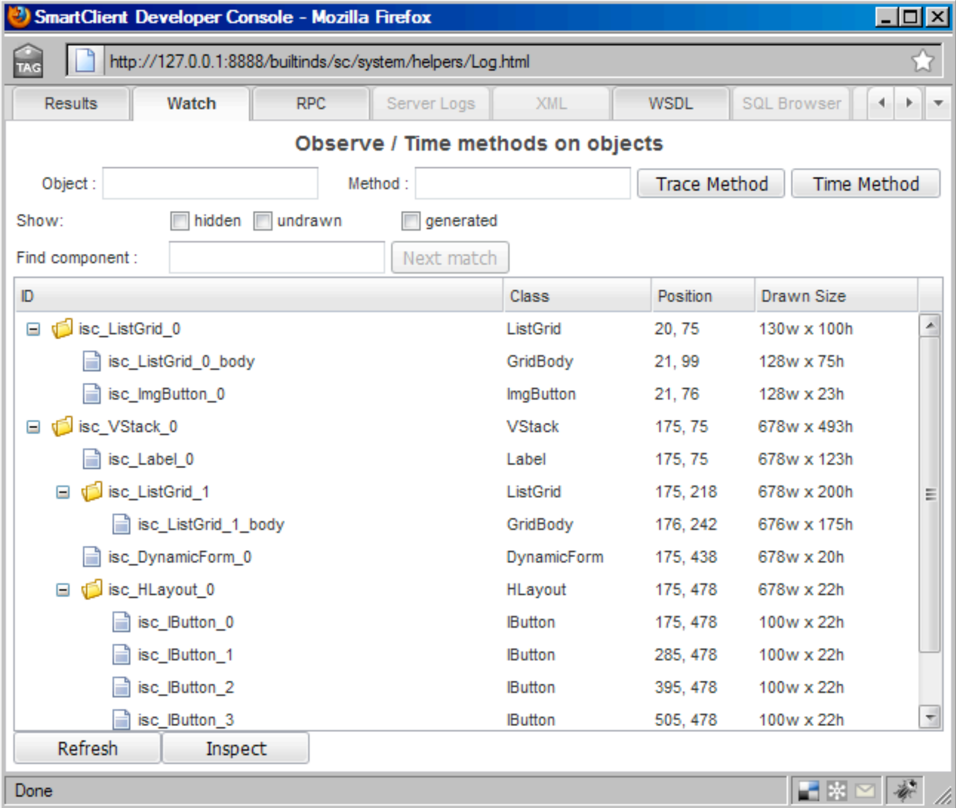
Popup blocker utilities may prevent the Developer Console from appearing, or browser security settings may disallow JavaScript URLs. Holding the **Ctrl** key while opening the console will bypass most popup blockers. Creating a bookmark for a JavaScript URL will allow you to execute it by clicking on the bookmark.

The **Results** tab of the Developer Console provides:

- Diagnostic messages logged by Smart GWT or your application code through the Smart GWT logging system. The *Logging Preferences* menu allows you to enable different levels of diagnostics in over 30 categories, from Layout to Events to Data Binding.

- Smart GWT component statistics. As you move the mouse in the current application, the ID of the current component under the mouse pointer is displayed in this area.

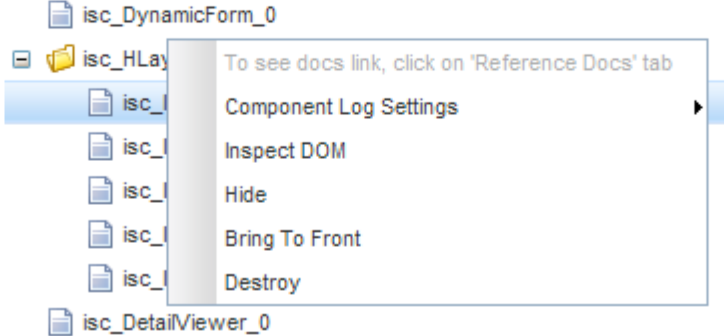
The **Watch** pane of the Developer Console displays a tree of Smart GWT user interface components in the current application. With the built-in-ds application running, this pane appears as follows:



In the **Watch** tab, you may:

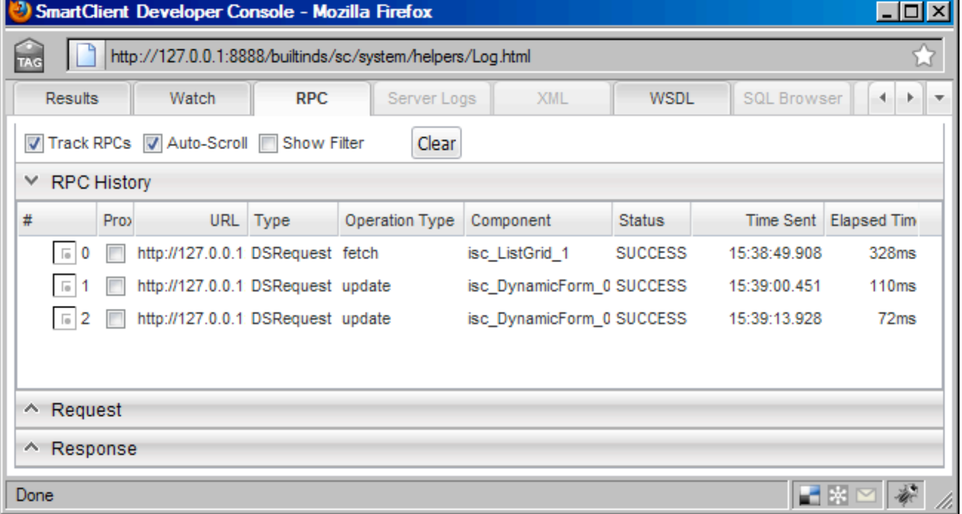
- Click on any item in the tree to highlight the corresponding component in the main application window with a flashing, red-dotted border.

- Right-click on any item in the tree for a menu of operations on that component.



- Click on the “hidden,” “undrawn,” and “generated” checkboxes to reveal additional components which are not currently visible, or are internally generated subcomponents of the Smart GWT components you have created.

The **RPC** tab shows requests for data issued by Smart GWT components.



Enable this tab by checking the “Track RPCs” checkbox. This tool shows you:

- Which component issued the request
- What logical type of request it was (such as a DataSource “update” request)
- A logical view of the request and response objects (instead of the raw HTTP request)

The Developer Console is an essential tool for all Smart GWT application developers and should be open whenever you are working with Smart GWT. For easy access, you should create a toolbar link to quickly show the Console:

In Firefox/Mozilla:

1. Show your Bookmarks toolbar if it is not already visible (*View* > *Toolbars* > *Bookmarks Toolbar*).

2. Go to the Bookmarks menu and pick *Manage Bookmarks…*

3. Click the *New Bookmark* button and enter "javascript:isc.showConsole()" as the bookmark Location, along with whatever name you choose.

4. Drag the new bookmark into the Bookmarks Toolbar folder.

In Internet Explorer:

1. Show your Links toolbar if it is not already visible (*View* > *Toolbars* > *Links*).

2. Type "javascript:isc.showConsole()" into the Address bar.

3. Click on the small Isomorphic logo in the Address bar and drag it to your Links toolbar.

4. If a dialog appears saying "You are adding a favorite that may not be safe. Do you want to continue?" click Yes.

5. If desired, rename the bookmark ("isc" is chosen as a default name)

The Developer Console is associated with a single web browser tab or window at any time. If you have shown the console for a Smart GWT application in one browser window, and then open an application in another browser window, you must close the console before you can show it from the new window.

Community Wiki

The wiki is a place where community members and Isomorphic engineers can work together to add new documentation and new samples - especially samples and documentation involving third-party products, where it would not be possible to create a standalone running sample in the SDK.

This is the right place to look for articles on how to integrate with a specific authentication system, application server, reporting engine, widget kit, cloud provider or similar major third-party product.

Isomorphic also publishes example code on the wiki before it becomes an official product sample, or is incorporated as a new product feature.

http://wiki.smartclient.com/

FAQ

The Smart GWT FAQ is maintained in the Forums, and contains up-to-the-minute information about recurring issues users are encountering, especially issues and advice regarding third-party tools or errors that may be encountered when straying from best practices.

http://forums.smartclient.com/showthread.php?t=8159

4. Visual Components

Smart GWT provides two families of visual components for rich web applications:

- ***Independent visual components***, which you create and manipulate directly in your applications.
- ***Managed form controls***, which are created and managed automatically by their "parent" form or editable grid.

This chapter provides basic usage information for the independent components only. Managed form controls are discussed in more detail in Chapter 5, *Data Binding*, and especially Chapter 6, *Layout*.

Component Documentation & Examples

Visual components encapsulate and expose most of the public capabilities in Smart GWT, so they have extensive documentation and examples in the Smart GWT SDK:

Smart GWT Java Doc—For component interfaces (APIs), see *http://www.smartclient.com/smartgwtee/javadoc/com/smartgwt/client/widgets/package-summary.html*.

Form controls are covered in*:* *http://www.smartclient.com/smartgwtee/javadoc/com/smartgwt/client/widgets/form/fields/package-summary.html*

Component Code Examples—For live examples of component usage, see the Smart GWT Showcase: *http://www.smartclient.com/smartgwt/showcase* . To view the code for these examples, click on the "View Source" button in the upper right corner of the tab.

The remainder of this chapter describes basic management and manipulation of ***independent visual components*** only. For

information on the creation and layout of managed form controls, see Chapter 5, *Data Binding*, and Chapter 6, *Layout*, respectively.

Drawing, Hiding, and Showing Components

To cause a Smart GWT component to draw (create its appearance via HTML), call `draw()`.

```
Label labelHello = new Label("Hello World");
labelHello.draw();
```

Components can subsequently be hidden and re-shown as a user navigates between different parts of the application, using these APIs:

- `hide()`
- `show()`

For example, to hide the label that was just drawn:

```
labelHello.hide();
```

UI components built into GWT itself (under the `com.google.gwt` package) are typically made visible by calling `RootPanel.get().add(widget)`. This is **not** the recommend approach for Smart GWT widgets. See Chapter 10, *Tips*, for more information.

Size and Overflow

The most basic properties for a visual component involve its size and overflow:

- `width`
- `height`
- `overflow`

Width and height allow either integer values, representing a number of pixels, or percentage values expressed as a String (such as `"50%"`), representing a percentage of the container's size (the entire web page, or another Smart GWT component). For example:

```
Label labelHello = new Label ("Hello World");
labelHello.setWidth(10);
```

In this example, the specified width is smaller than the contents of the label, so the text wraps and “overflows” the specified size of the label. This behavior is controlled by the overflow property, which is managed automatically by most components. You may want to change this setting for `Canvas`, `Label`, `DynamicForm`, `DetailViewer`, or `Layout` components, whose contents you want to clip or scroll instead. To do this, set the overflow property to “hidden” (clip) or “auto” (show scrollbars when needed). For example:

```
import com.smartgwt.client.types.Overflow;

Label labelHello = new Label ("Hello World");
labelHello.setWidth(10);
labelHello.setOverflow(Overflow.HIDDEN);
```

This will show a 10 pixel wide Label for which the text “Hello World” is clipped.

In most applications, you will place your components into layout managers which dynamically determine their size and position. Chapter 6, *Layout*, introduces the Smart GWT Layout managers, which you can use to automatically size, position, and reflow your components at runtime.

Handling Events

Smart GWT applications implement interactive behavior by responding to *events* generated by their environment or user actions. You can provide the logic for hundreds of different events by implementing event *handlers*.

The most commonly used Smart GWT events include:

- `Click` (for buttons and menu items)
- `RecordClick` (for list grids and tree grids)
- `Changed` (for form controls)
- `TabSelected` (for tabsets)

If a Smart GWT component has support for a given type of event, it will implement a Java interface `Has`*`EventName`*`Handlers` such as `HasClickHandlers`. These interfaces allow registration of an `EventHandler` object that receives an `Event` object when the event occurs.

For example:

```
import com.smartgwt.client.widgets.Button;

Button btn = new Button("Click me");
btn.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        // respond to the event here
    }
});
```

The `Event` object your event handler receives has methods that allow you to retrieve information about the current event. For example, the `RecordClickEvent` has a `getRecord()` method that allows you to retrieve the `Record` that was clicked.

In addition to methods on `Event` objects, information common to many types of events is available from static methods on the central `EventHandler` class in the package `com.smartgwt.client.util`.

For more information on available Smart GWT events, see:

- *Smart GWT JavaDoc*—Component-specific APIs under *com.smartgwt.client.widgets*.

5. Data Binding

Databound Components

You can bind certain Smart GWT components to *DataSources*, which provide their structure and contents. The following visual components are designed to display, query, and edit structured data:

Visual Component	Query & Display Data	Edit Data
`DynamicForm`	✓	✓
`ListGrid`	✓	✓
`TreeGrid`	✓	✓
`Calendar`	✓	✓
`DetailViewer`	✓	
`TileGrid`	✓	
`ColumnTree`	✓	

Databound components provide you with both automatic and manual databinding behaviors. For example:

- *Automatic behavior*—A databound `ListGrid` will generate `Fetch` operations when a user scrolls the list to view more records.

- *Manual behavior*—You can call `removeSelectedData()` on a databound `ListGrid` to perform `Remove` operations on its datasource.

This chapter outlines the *client-side* interfaces that you may use to configure databound components and interact with their underlying DataSources. Chapter 7, *Data Integration*, outlines the interfaces

for *server-side* integration of DataSources with your data and service tiers.

Fields

Fields are the building blocks of databound components and DataSources. There are two types of field definitions:

- ***Component*** fields provide ***presentation*** attributes for databound visual components (such as width and alignment of columns in a table). Component fields are discussed immediately below.

- ***DataSource*** fields provide ***metadata*** describing the objects in a particular datasource (such as data type, length, and required).

Component fields display as the following sub-elements of your databound components:

Component	Fields
`DynamicForm`	form controls
`ListGrid`	columns & form controls
`TreeGrid`	columns & form controls
`Calendar`	event duration and description
`ColumnTree`	columns & form controls
`DetailViewer`	rows within blocks
`TileGrid`	rows within tiles
`CubeGrid` (Analytics option)	facets (row & column headers)

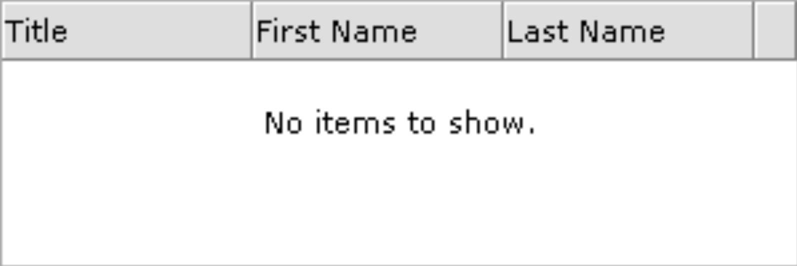
You can specify the displayed fields of a visual component via the `setFields()` method, which takes an array of objects defining the fields for that component. For example:

```
ListGrid grid = new ListGrid();

ListGridField salutationField = new ListGridField();
salutationField.setName("salutation");
salutationField.setTitle("Title");

ListGridField firstNameField = new ListGridField();
firstNameField.setName("firstname");
firstNameField.setTitle("First Name");

ListGridField lastNameField = new ListGridField();

lastNameField.setName("lastname");
lastNameField.setTitle("Last Name");

grid.setFields(salutationField,
               firstNameField,
               lastNameField);

grid.draw();
```

Note that `ListGridField` actually has shortcut constructors that simplify this, for example:

```
ListGridField salutationField =
    new ListGridField ("salutation", "Title");
```

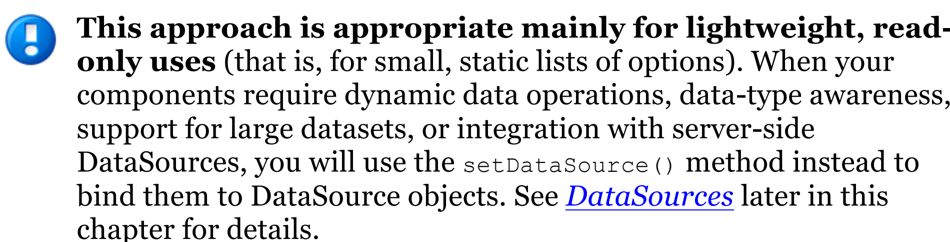
Try running the example code above. When you load it in your web browser, you should see a `ListGrid` that looks like this:



The `name` property of a field is the special key that connects that field to actual data values. To add records to the fields, add this code to the `ListGrid` definition above:

```
ListGridRecord record1 = new ListGridRecord();
record1.setAttribute("salutation", "Ms");
record1.setAttribute("firstname","Kathy");
record1.setAttribute("lastname","Whitting");

ListGridRecord record2 = new ListGridRecord();
record2.setAttribute("salutation", "Mr");
record2.setAttribute("firstname","Chris");
record2.setAttribute("lastname","Glover");

ListGridRecord record3 = new ListGridRecord();
record3.setAttribute("salutation", "Mrs");
record3.setAttribute("firstname","Gwen");
record3.setAttribute("lastname","Glover");

grid.setData(new ListGridRecord[] {
    record1, record2, record3
});
```

Now when you load this example, you should see:



Title	First Name	Last Name
Ms	Kathy	Whitting
Mr	Chris	Glover
Mrs	Gwen	Glover

This approach is appropriate mainly for lightweight, read-only uses (that is, for small, static lists of options). When your components require dynamic data operations, data-type awareness, support for large datasets, or integration with server-side DataSources, you will use the `setDataSource()` method instead to bind them to DataSource objects. See *DataSources* later in this chapter for details.

The basic properties of field definitions (`name`, `title`) in the ListGrid above are the same as the properties used across other components that support data binding. For example, this very similar code creates a `DynamicForm` for working with the same `Records`:

```
import com.smartgwt.client.widgets.form.fields.TextItem;
import com.smartgwt.client.widgets.form.fields.FormItem;

DynamicForm form = new DynamicForm();

TextItem salutationItem = new TextItem();
salutationItem.setName("salutation");
salutationItem.setTitle("Title");

TextItem firstNameItem = new TextItem();
firstNameItem.setName("firstname");
firstNameItem.setTitle("First Name");

TextItem lastNameItem = new TextItem();
lastNameItem.setName("lastname");
lastNameItem.setTitle("Last Name");

form.setFields(new FormItem[] {
      salutationItem, firstNameItem, lastNameItem
});

form.draw();
```

This will display as:

Title :
First Name :
Last Name :

For complete documentation of field properties (presentation attributes) for `ListGrid` and `DynamicForm`, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.widgets.form.fields

- Smart GWT JavaDoc:

  com.smartgwt.client.widgets.grid.ListGridField

DataSources

Smart GWT *DataSource* objects provide a presentation-independent, implementation-independent description of a set of persistent data fields. DataSources enable you to:

- Share your data models across multiple applications and components, and across both client and server.
- Display and manipulate persistent data and data-model relationships (such as parent-child) through visual components (such as `TreeGrid`).
- Execute standardized data operations (`fetch`, `sort`, `add`, `update`, `remove`) with built-in support on both client and server for data typing, validators, paging, unique keys, and more.
- Leverage automatic behaviors including data loading, caching, filtering, sorting, paging, and validation.

A DataSource *descriptor* provides the attributes of a set of DataSource fields. DataSource descriptors can be specified in XML format or created in Java code. The XML format is interpreted and shared by both client and server, while DataSources created in Java are used by the client only.

Note that use of the XML format requires the Smart GWT Server. The Smart GWT Server can also derive DataSources from existing sources of metadata formats such as Java Beans or SQL schema – see Chapter 8, *Smart GWT Server Framework*, for details.

There are four basic rules to creating XML DataSource descriptors:

1. Specify a unique DataSource `ID` attribute. The ID will be used to bind to visual components, and as a default name for object-relational (table) bindings and test data files.
2. Specify a field element with a unique name (unique within the DataSource) for each field that will be used in a databound UI component.
3. Specify a type attribute on each field element (see the code sample that follows for supported data types).
4. Mark a field with `primaryKey="true"`. The `primaryKey` field must have a unique value in each data object (record) in a DataSource. A `primaryKey` field is not required for read-only DataSources, but it is a good general practice to allow for future `add`, `update`, or `remove` data operations. If you need multiple primary keys, see Chapter 10, *Tips*.

Following these rules, a DataSource descriptor for personal contacts might look as follows:

```
<DataSource ID="contacts">
<fields>
    <field primaryKey="true"
          name="id"          hidden="true"      type="sequence" />
    <field name="salutation" title="Title"      type="text" >
        <valueMap>
          <value>Ms</value>
          <value>Mr</value>
          <value>Mrs</value>
        </valueMap>
        </field>
    <field name="firstname"  title="First Name" type="text" />
    <field name="lastname"   title="Last Name"  type="text" />
    <field name="birthday"   title="Birthday"   type="date" />
    <field name="employment" title="Status"     type="text" >
        <valueMap>
          <value>Employed</value>
          <value>Unemployed</value>
        </valueMap>
        </field>
    <field name="bio"        title="Bio"        type="text"
                                                length="2000" />
    <field name="followup"   title="Follow up"  type="boolean" />
</fields>
</DataSource>
```

To load this DataSource:

5. Save the XML as `contacts.ds.xml` in your project, under `war/ds`. This is the default location that the Smart GWT server framework looks for DataSource descriptors.

6. Add a `<script>` tag to your HTML bootstrap file that uses the `DataSourceLoader` servlet to load the DataSource. For example, in the "built-in-ds" sample project, the bootstrap file is `war/BuiltinDS.html`, and there is an existing `<script>` tag you can add to. Change it to:

    ```
    <script src="builtinds/sc/DataSourceLoader?dataSource=supplyItem,
    animals,employees,contacts"></script>
    ```

    In your Java code, access the DataSource like this:

    ```
    DataSource contactsDS = DataSource.get("contacts");
    ```

    You can now supply this DataSource to the components via the `setDataSource()` method. The complete code for a page that binds a grid and form to this DataSource is:

    ```
    DataSource contactsDS = DataSource.get("contacts");

    ListGrid grid = new ListGrid();
    grid.setDataSource(contactsDS);

    DynamicForm form = new DynamicForm();
    form.setLeft(300); // avoid overlap
    form.setDataSource(contactsDS);
    ```

In this example, the `grid` and `form` components are now automatically generating component fields based on DataSource fields. Note that the form has chosen specific controls for certain fields—it does so based on the following rules:

Field attribute	Form control
`valueMap` provided	`SelectItem` (dropdown)
type `boolean`	`CheckboxItem` (checkbox)
type `date`	`DateItem` (date control)
`length` > 255	`TextAreaItem` (large text box)

You can override these choices by setting the `editorType` attribute on the `<field>` tag in the `.ds.xml` file to the Java classname of the desired `FormItem`.

Other common DataSource field properties include:

Property	Values
`name`	Unique field identifier (required on every DataSource field)
`type`	"`text`" | "`integer`" | "`float`" | "`boolean`" | "`date`" | "`datetime`" | "`time`" | "`enum`" | "`sequence`" | "`binary`" See reference for full list of field types.
`title`	Default user-visible label for the field.
`length`	Maximum length of text value in characters.
`hidden`	specifies whether this field should be hidden from the end user. When hidden, it will not appear in the default presentation, and it will not appear in any field selectors (such as the column picker menu in a `ListGrid`).
`detail`	Defaults to true; specifies whether this field is a "detail" that should not be shown by default in multi-record summary views such as a `ListGrid`.
`required`	"`true`" | "`false`"; applies validation on both client and server to verify that the field is non-blank.
`valueMap`	An array of values, or an object containing `storedValue:displayValue` pairs.
`editorType`	`FormItem` class to use when editing this field (in any `DataBoundComponent`).

`primaryKey`	specifies whether this is the field that uniquely identifies each record in this DataSource (that is, it must have a unique value for each record). The `primaryKey` field is often specified with `type="sequence"` and `hidden="true"`, to generate a unique internal key. For multiple primary keys, see Chapter 10, *Tips*.
`foreignKey`	A reference to a field in another DataSource (`dsName.fieldName`).
`rootValue`	For fields that establish a tree relationship (by `foreignKey`), this value indicates the root node of the tree.

For more information on DataSources and a complete reference of properties that may be set in `.ds.xml` files, see

- Smart GWT JavaDoc:

  com.smartgwt.client.docs.serverds

  Do not import this package—it is informational only.

For DataSource usage examples, see the descriptors in `samples\showcase\war\ds`. These DataSources are used in various Smart GWT SDK examples, including the Smart GWT EE Showcase.

For an example of a DataSource relationship using `foreignKey`, see the `TreeGrid` example in the Smart GWT EE Showcase (TreeGrid UI) and `samples\showcase\war\ds\employees.ds.xml` (associated DataSource).

Customized Data Binding

You can also *combine* DataSource binding *and* component-specific field definitions. Smart GWT merges your component field definitions and DataSource field definitions by using the `name` property of the fields to match component fields with DataSource fields.

In this case, component field definitions specify presentation attributes specific to that component, while the DataSource field definitions specify data attributes and presentation attributes common to all `DataBoundComponent`s.

By combining component-specific fields and DataSource fields, you can share data model attributes and common presentation attributes across all components, while still allowing full customization of individual components for a specific use case. For example, a `ListGrid` component might specify a shorter `title` attribute for more compact display ("Percentage" represented as "%").

Components can also have additional fields not defined in the DataSource. For example, a user registration form might have a second password field to ensure that the user has typed his password correctly.

By default, a component with both fields and a DataSource will only show the fields defined on the component, in the order they are defined on the component. To change this behavior, use `setUseAllDataSourceFields(true)`. Now, all DataSource fields will be shown unless you provide a component field definition where you have called `setHidden(true)`.

For an example of customized binding, see Forms → Validation → Customized Binding in the Smart GWT Showcase.

For more information on the *layout* of managed form controls, see the section *Form Layout* in Chapter 6.

DataSource Operations

Smart GWT provides a standardized set of data operations that act upon DataSources:

Operation	Methods	Description
Fetch	`fetchData(…)`	Retrieves records from the datasource that match the provided criteria.
Add	`addData(…)`	Creates a new record in the datasource with the provided values.
Update	`updateData(…)`	Updates a record in the datasource with the provided values.
Remove	`removeData(…)`	Deletes a record from the datasource that exactly matches the provided criteria.

These methods each take three parameters:

- a **data** object containing the criteria for a Fetch or Filter operation, or the values for an Add, Update, or Remove operation
- a **callback** expression that will be evaluated when the operation has completed
- a **properties** object containing additional parameters for the operation—timeout length, modal prompt text, and so on (see `DSRequest` in the *Smart GWT Reference* for details)

You may call any of these four methods directly on a DataSource object or on a databound `ListGrid` or `TreeGrid`.

For example, the following code saves a new Record to the `contactDS` DataSource and displays a confirmation when the save completes:

```
import com.smartgwt.client.data.DSCallback;
import com.smartgwt.client.data.DSResponse;

Record contact = new Record();
contact.setAttribute("salutation", "Mr.");
contact.setAttribute("firstname", "Steven");
contact.setAttribute("lastname", "Hudson");

DSCallback callback = new DSCallback() {
    public void execute(DSResponse response,
                        Object rawData,
                        DSRequest request)
    {
        Record savedContact = response.getData()[0];
        SC.say(savedContact.getAttribute("firstname")+
               "added to contact list");
    }
};

contactDS.addData(contact,callback);
```

DataSource operations will only execute successfully if the DataSource is bound to a persistent data store. You can create relational database tables from a DataSource `.ds.xml` file by using the *Import DataSources* section in the Smart GWT Admin Console. For integration with pre-existing business logic or non-SQL persistence systems, see Chapter 7, *Data Integration*.

DataBound Component Operations

In addition to the standard DataSource operations listed above, you can perform Add and Update operations from databound form components by calling the following DynamicForm methods:

Method	**Description**
`editRecord()`	Starts editing an existing record
`editNewRecord()`	Starts editing a new record
`saveData()`	Saves the current edits (add new records; update existing records)

Databound components also provide several convenience methods for working with the selected records in components that support selection, such as ListGrid:

Convenience Method
listGrid.**removeSelectedData**()
dynamicForm.**editSelectedData**(*listGrid*)
detailViewer.**viewSelectedData**(*listGrid*)

Each sample in the `samples` directory in the SDK shows the most common DataBoundComponents interacting with DataSources.

Data Binding Summary

This chapter began by introducing Databound Components, to build on the concepts of the previous chapter, *Visual Components*. However, in actual development, DataSources usually come first. The typical steps to build a databound user interface with Smart GWT components are:

1. Create DataSource descriptors (`.ds.xml` files), specifying data model (metadata) properties in the DataSource fields.

2. Back your DataSources with an actual data store. The Smart GWT Admin Console GUI can create and populate relational database tables for immediate use. Chapter 7, *Data Integration*, describes the integration points for binding to other object models and data stores.

3. Load DataSource descriptors in your Smart GWT-enabled pages using a standard `<script src=…>` HTML tag referencing the DataSourceLoader servlet. Or, for DataSources that do not use the Smart GWT server, create them programmatically in Java.

4. Create and bind visual components to DataSources using the `setDataSource()` method with components that support data-binding.

5. Modify component-specific presentation properties by adding customized field definitions where necessary

6. Call databound component methods (such as `fetchData`) to perform standardized data operations through your databound components.

DataSources effectively hide the back-end implementation of your data and service tiers from your front-end presentation—so you can change the back-end implementation at any time, during development or post-deployment, without changing your client code.

See Chapter 7, *Data Integration*, for an overview of server-side integration points that address all stages of your application lifecycle.

6. Layout

Component Layout

Most of the code snippets seen so far create just one or two visual components, and position them manually with the `left`, `top`, `width`, and `height` properties.

This manual layout approach becomes brittle and complex with more components. For example, you may want to:

- allocate available space based on relative measures (such as 30%)
- resize and reposition components when other components are resized, hidden, shown, added, removed, or reordered
- resize and reposition components when the browser window is resized by the user

Smart GWT includes a set of *layout managers* to provide these and other automatic behaviors. The Smart GWT layout managers implement consistent dynamic sizing, positioning, and reflow behaviors that cannot be accomplished with HTML and CSS alone.

The fundamental Smart GWT layout manager is implemented in the Layout class, which provides four subclasses to use directly:

- `HLayout` manages the positions and widths of a list of components in a horizontal sequence.
- `VLayout` manages the positions and heights of a list of components in a vertical sequence.
- `HStack` positions a list of components in a horizontal sequence, but does not manage their widths.
- `VStack` positions a list of components in a vertical sequence, but does not manage their heights.

These layout managers are, themselves, visual components, so you can create and configure them the same way you would create a `Label`, `Button`, `ListGrid`, or other independent component.

The main properties of a layout manager are:

Layout property	Description
`members`	An array of components managed by this layout.
`membersMargin`	Number of pixels of space between each member of the layout.
`layoutMargin`	Number of pixels of space surrounding the entire layout.

The member components also support additional property settings in the context of their parent layout manager:

Member property	Description
`layoutAlign`	Alignment with respect to the breadth axis of the layout ("left," "right," "top," "bottom," or "center").
`showResizeBar`	Specifies whether a drag-resize bar appears between this component and the next member in the layout. ("true" | "false").
`width`, `height`	Layout-managed components support a "*" value (in addition to the usual number and percentage values) for their size on the length axis of the layout, to indicate that they should take a share of the remaining space after fixed-size components have been counted (this is the default behavior if no width/height is specified).

Components that automatically size to fit their contents will not be resized by a layout manager. By default, `Canvas`, `Label`, `DynamicForm`, `DetailViewer`, and `Layout` components have `set.Overflow(Overflow.VISIBLE)`, so they expand to fit their contents. If you want one of these components to be sized by a layout instead, you must set its overflow property to hidden (clip) or auto (show scrollbars when needed).

Layout managers may have other layout managers as members. By nesting combinations of `HLayout` and `VLayout`, you can create complex

dynamic layouts that would be difficult or impossible to achieve in HTML and CSS.

You can use the special `LayoutSpacer` component to insert extra space into your layouts. For example, here is the code to create a basic page header layout, with a left-aligned logo and right-aligned title:

```
import com.smartgwt.client.widgets.Img;
import com.smartgwt.client.widgets.layout.LayoutSpacer;

HLayout hLayout = new HLayout(10);
hLayout.setID("myPageHeader");
hLayout.setHeight(50);
hLayout.setLayoutMargin(10);
hLayout.addMember(new Img("myLogo.png"));
hLayout.addMember(new LayoutSpacer());
hLayout.addMember(new Label("My Title"));
hLayout.draw();
```

Container Components

In addition to the basic layout managers, Smart GWT provides a set of rich container components. These include:

`SectionStack`	to manage multiple stacked, user-expandable and collapsible 'sections' of components
`TabSet`	to manage multiple, user-selectable 'panes' of components in the same space
`Window`	to provide free-floating, modal and non-modal views that the user can move, resize, maximize, minimize, or close

See the Smart GWT Demo Application (*http://www.smartclient.com/smartgwt/showcase/#featured_complete_app*) for examples of various layout components in action.

For more information, see *com.smartgwt.client.widgets.layout*

Form Layout

Data entry forms have special layout requirements—they must present their controls and associated labels in regularly aligned rows and columns, for intuitive browsing and navigation.

When form controls appear in a `DynamicForm`, their positions and sizes are controlled by the Smart GWT *form layout manager*. The form layout manager generates a layout structure similar to an HTML table. Form controls and their titles are rendered in a grid from left-to-right, top-to-bottom. You can configure the high-level structure of this grid with the following `DynamicForm` properties:

DynamicForm property	Description
`numCols`	Total number of columns in the grid, for form controls and their titles. Set to a multiple of 2, to allow for titles, so `numCols:2` allows one form control per row and `numCols:4` allows two form controls per row.
`titleWidth`	Number of pixels allocated to each title column in the layout.
`colWidths`	Optional array of pixel widths for all columns in the form. If specified, these widths will override the column widths calculated by the form layout manager.

You can control the positioning and sizing of form controls in the layout grid by changing their positions in the fields array, their `height` and `width` properties, and the following field properties:

Field property	Description
`colSpan`	Number of form layout columns occupied by this control (not counting its title, which occupies another column)
`rowSpan`	Number of form layout rows occupied by this control
`startRow`	Specifies whether this control should always start a new row. (“true” | “false”)
`endRow`	Specifies whether this control should always end its row. (“true” | “false”)

Field property	Description
`showTitle`	Specifies whether this control should display its title. (“true” | “false”)
`align`	Horizontal alignment of this control within its area of the form layout grid (“left,” “right,” or “center”)

See *Showcase* → *Forms* → *Form Layout* for examples of usage of these properties.

`ButtonItem` has both `startRow:true` and `endRow:true` by default. To place a button next to a text field or other form component, one or both of these properties must be set false, and enough columns must exist for both the button and any controls it is adjacent to (for example, 3 for a `TextItem` with title and a `ButtonItem`).

You can also use the following special form items to include extra space and formatting elements in your form layouts:

- `HeaderItem`
- `BlurbItem`
- `SpacerItem`
- `RowSpacerItem`

See the JavaDoc for these classes for properties that can be set for additional control.

For more information on form layout capabilities, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.docs.FormLayout

- Smart GWT JavaDoc:

  com.smartgwt.client.widgets.form.fields.FormItem

7. Data Integration

Smart GWT DataSources provide a data-provider-agnostic interface to databound components, allowing those components to implement sophisticated behaviors that can be used with any data provider. In this chapter, we explain how to integrate a DataSource with various persistence systems so that the operations initiated by databound components can retrieve and modify persistent data.

DataSource Requests

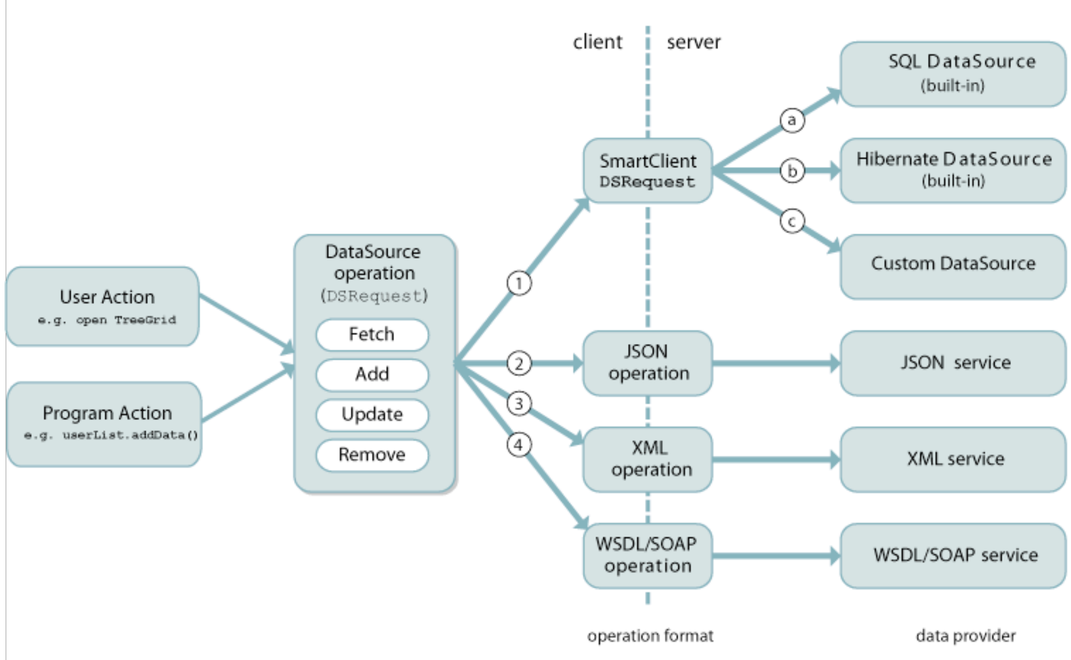
When a visual component, or your own custom code, attempts to use a DataSource operation, a `DSRequest` (DataSource Request) is created representing the operation. "Data Integration" is the process of fulfilling that `DSRequest` by creating a corresponding `DSResponse` (DataSource Response), by using a variety of possible approaches to connect to the ultimate data provider.

There are two main approaches to fulfilling DataSource Requests:

- **Server-side integration (Smart GWT Server Framework)**: DataSource requests from the browser arrive as Java Objects on the server. You deliver responses to the browser by returning Java Objects. This is the simpler and more powerful approach.

- **Client-side integration**: DataSource requests arrive as HTTP requests which your server code receives directly (in Java, you use the Servlet API or `.jsp` files to handle the requests). Responses are sent as XML or JSON, which you directly generate.

The possible approaches to data integration are summarized in the following diagram. Paths 2, 3 and 4 are client-side integration approaches, while path 1 includes all server-side integration approaches.



Smart GWT Server Framework

Path 1 makes use of the Smart GWT Server Framework. Available with Pro edition and above, the server framework is a set of Java libraries and servlets that can be integrated with any pre-existing Java application.

Unless you are forced to use a different approach (for example, you are not using a Java-based server, and cannot deploy a Java-based server in front of your existing server), it is **highly** recommended that you use the Smart GWT Server Framework for data integration. The server framework delivers an immense range of functionality that compliments any existing application and persistence engine. Chapter 8, *Smart GWT Server Framework*, provides an overview.

If you cannot use the Smart GWT Server Framework, the best approaches for data integration are covered later in this chapter.

DSRequests and DSResponses

Regardless of the data integration approach used, the data in the request and response objects has the same meaning.

The key members of a `DSRequest` object are:

- `data`: the search criteria (for "fetch"), new record values ("add" or "update") or criteria for the records to delete ("remove")
- `sortBy`: requested sort direction for the data ("fetch" only)
- `startRow` and `endRow`: the range of records to fetch (if paging is active)
- `oldValues`: values of the record before changes were made, for checking for concurrent edits (all operations but "fetch")

The key members of a `DSResponse` object are:

- `status`: whether the request succeeded or encountered a validation or other type of error
- `data`: the matching records (for "fetch"), data-as-saved ("add" or "update"), or deleted record ("remove")
- `startRow` and `endRow`: the range of records actually returned (if paging is active)
- `totalRows`: the total number of records available
- `errors`: for a validation error response, fields that were invalid and error messages for each

Request and Response Transformation

If you are using the Smart GWT Server Framework with one of the built-in DataSource types (such as SQL or JPA/Hibernate), you will not need to do any request or response transformation work and can proceed directly to Chapter 8, *Smart GWT Server Framework*.

If you cannot use the server framework, but you are free to define the format and content of messages passed between the browser and your server, the simplest data integration approach is the `RestDataSource` class.

The `RestDataSource` performs the four core DataSource operations using a simple, well-documented protocol of XML or JSON requests and responses sent over HTTP. The HTTP requests sent by the client will contain the details of the `DSRequest` object and your server-side code

should parse this request and output an XML or JSON formatted response containing the desired properties for the `DSResponse`.

If, instead, you are required to integrate with a pre-existing service that defines its own HTTP-based protocol, you can configure a subclass of the `DataSource` class to customize the HTTP request format and the expected format of responses.

For services that return XML or JSON data (including WSDL), you can specify XPath expressions indicating what part of the data should be transformed into `dsResponse.data`. If XPath expressions are not sufficient, you can override `DataSource.transformRequest()` and `DataSource.transformResponse()` and add Java code to handle those cases.

These same two APIs (`transformRequest` and `transformResponse`) enable integration with formats other than XML and JSON, such as CSV over HTTP or proprietary message formats.

Finally, setting `DataSource.dataProtocol` to `DSProtocol.CLIENTCUSTOM` prevents a DataSource from trying to directly send an HTTP request, allowing integration with data that has already been loaded by a third party communication system, or integration in-browser persistence engines such as HTML5 localStorage or in-browser SQLite databases.

To learn more about using the `RestDataSource` and client-side data integration options, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.data.RestDataSource

- Smart GWT JavaDoc:

  com.smartgwt.client.docs.ClientDataIntegration

For a live sample of `RestDataSource` showing sample responses, see:

- Smart GWT Showcase

  showcase/#featured_restfulds

Criteria, Paging, Sorting and Caching

Smart GWT UI components such as the `ListGrid` provide an interface that allows an end user to search data, sort data, and page through large datasets. As this interface is used, the UI component generates `DSRequests` that will contain search criteria, requested sort directions and requested row ranges.

However, Smart GWT does not *require* that a data provider implement all of these capabilities. In fact, Smart GWT is able to use a "flat file" as a response to the "fetch" operation, and implement searching and sorting behaviors within the browser.

If a data provider cannot implement paging and sorting behaviors, it is sufficient to simply ignore the `startRow`, `endRow` and `sortBy` attributes of the `DSRequest` and return a `DSResponse` containing all data that matches the provided criteria, in any order. Smart GWT will perform sorting client-side as necessary. This does not need to be configured in advance – a data provider can decide, on a case-by-case basis, whether to simply return all data for a given request.

If a data provider also cannot implement the search behavior, the DataSource can be set to `cacheAllData`. This means that the first time any data is requested, all data will be requested (specifically, a `DSRequest` will be sent with no search criteria). Smart GWT will then perform searches within the browser. Data modification requests ("add", "update" or "remove" operations) are still sent normally – in effect, a "write-through" cache is maintained.

To learn more about searching, sorting, paging and caching behaviors, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.data.ResultSet

  com.smartgwt.client.data.DataSource#cacheAllData

Authentication and Authorization

Securing Smart GWT applications is done in substantially the same way as standard web applications. In fact, Smart GWT's advanced architecture actually simplifies the process and makes security auditing easier.

For example, enabling HTTPS requires no special configuration. Simply ensure that any URLs provided to Smart GWT do not include an explicit "http://" at the beginning, and all `DSRequests`, requests for images and so forth will automatically use the "https://" prefix and be protected.

Although it is straightforward to build a login interface in Smart GWT, it is generally recommended that you implement your login page as a plain HTML page, due to the following advantages:

- interoperable/single sign-on capable —if your application may need to participate in single sign-on environment (even in the distant future), you will be in a better position to integrate if you are making minimal assumptions about the technology and implementation of the login page

- login page appears instantly—the user does not have to wait for the entire application to download in order to see the login page and begin entering credentials

- background loading – use techniques such as off-screen <img> tags and <script defer=true/> tags to begin loading your application while the user is typing in credentials

Most authentication systems feature the ability to protect specific URLs or URLs matching a pattern or regular expression, such that a browser will be redirected to a login page or given an access denied error message. When securing your Smart GWT application:

- **Do** protect the URL of your bootstrap HTML file. Unauthenticated users should be redirected to the login page when this URL is accessed.

- **Do** protect the URLs that return dynamic data, for example, `sc/IDACall` if you are using the Smart GWT Server Framework, or the URL(s) you configure as `DataSource.dataURL` if not.

- **Do not** protect the static resources that are part of the skin or the SmartClient runtime underlying Smart GWT, specifically the URL patterns `sc/skins/*` and `sc/system/*`. These are publically available files; protecting them just causes a performance hit and in some cases can negatively affect caching

- **Consider** leaving GWT-generated application logic such as `*.cache.html` files unprotected. These files are heavily obfuscated and analyzing them is not a likely approach for an attacker to take. As with other static resources, not protecting these files provides a performance boost.

If you are using the Smart GWT Server Framework, see the *Declarative Security* section of Chapter 8 for further authentication and authorization features, including the ability to declare role-based security restrictions in `.ds.xml` file, create variations on DataSource operations accessible to different user roles, and create certain operations accessible to unauthenticated users.

Relogin

When a user's session has expired and the user tries to navigate to a protected resource, typical authentication systems will redirect the user to a login page. With Ajax systems such as Smart GWT, this attempted redirect may happen in response to background data operations, such as a form trying to save. In this case, the form perceives the login page as a malformed response and displays a warning, and the login page is never displayed to the user.

The ideal handling of this scenario is that the form's attempt to save is "suspended" while the user re-authenticates, then is completed normally. Smart GWT makes it easy to implement this ideal handling through the *Relogin* subsystem.

To enable Smart GWT to detect that session timeout has occurred, a special marker needs to be added to the HTTP response that is sent when a user's session has timed out. This is called the `loginRequiredMarker`.

When this marker is detected, Smart GWT raises a `LoginRequired` event, automatically suspending the current network request so that it can be later resubmitted after the user logs back in.

To learn more about the `loginRequiredMarker` and Relogin, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.docs.Relogin

8. Smart GWT Server Framework

The Smart GWT server framework is a set of Java libraries and servlets that can be integrated with any pre-existing Java application.

The server framework allows you to rapidly connect Smart GWT visual components to pre-existing Java business logic, or can provide complete, pre-built persistence functionality based on SQL, Hibernate, JPA or other Java persistence frameworks.

DataSource Generation

The server framework allows you to generate DataSource descriptors (`.ds.xml` files) from Java Beans or SQL tables. This can be done as either a one-time generation step, or can be done *dynamically*, creating a direct connection from the property definitions and annotations on your Java Beans to your UI components.

This approach avoids the common problem of UI component definitions duplicating information that already exists on the server, while still enabling every aspect of data binding to be overridden and specialized for particular screens.

As an example, if you had the Java Bean `Contact`, the following is a valid DataSource whose fields would be derived from a Java Bean:

```
<DataSource ID="contacts" schemaBean="com.sample.Contact"/>
```

Using `schemaBean` doesn't imply any particular persistence engine; it uses the provided Java class for derivation of DataSource fields only.

The following DataSource would derive its fields from your database columns (as well as being capable of all CRUD operations):

```
<DataSource ID="contacts" serverType="sql"
            tableName="contacts" autoDeriveSchema="true" />
```

In many scenarios, an auto-derived DataSource is immediately usable for UI component databinding. Among other intelligent default behaviors, field titles appropriate for end users are automatically derived from Java property names and SQL column names by detecting common naming patterns.

For example, a Java property accessed by a method `getFirstName()` receives a default title of "First Name" by recognizing the Java "camelCaps" naming convention; a database column named `FIRST_NAME` also receives a default title of "First Name" by recognizing the common database column naming pattern of underscore-separated words.

The default rules for mapping between Java and SQL types and `DataSourceField` types are summarized in the following table:

Java Type	SQL JDBC Type	DataSource Field type
String, Character	CHAR, VARCHAR, LONGVARCHAR, CLOB	`text`
Integer, Long, Short, Byte, BigInteger	INTEGER, BIGINT, SMALLINT, TINYINT, BIT	`integer`
Float, Double, BigDecimal	FLOAT, DOUBLE, REAL, DECIMAL, NUMERIC	`float`
Boolean	<none>	`boolean`
Date, java.sql.Date	DATE	`date`
java.sql.Time	TIME	`time`
java.sql.Timestamp	TIMESTAMP	`datetime`
any Enum	<none>	`enum` (`valueMap` also auto-derived)

In addition to the Java types listed, primitive equivalents are also supported ("Integer" in the table above implies both `Integer` and `int`) as well as subclasses (for non-`final` types like Date).

You can customize the automatically generated fields in a manner similar to customizing the fields of a DataBound component. Fields declared with the same name as automatically derived fields will override individual properties from the automatically derived field; fields with new names will appear as added fields.

For example, you may have a database column `employment` that stores a one-character employment status code, and needs a `valueMap` to display appropriate values to end users:

```
<DataSource ID="contacts" serverType="sql"
            tableName="contacts" autoDeriveSchema="true">
    <fields>
        <field name="employment">
            <valueMap>
                <value ID="E">Employed</value>
                <value ID="U">Unemployed</value>
            </valueMap>
        </field>
    </fields>
</DataSource>
```

Field by field overrides are based on DataSource inheritance, which is a general purpose feature that allows a DataSource to inherit field definitions from another DataSource. In effect, `schemaBean` and `autoDeriveSchema` automatically generate an implicit parent DataSource. Several settings are available to control field order and field visibility when using DataSource inheritance, and these settings apply to automatically generated fields as well.

Finally, note that DataSource definitions are completely dynamic and actual .ds.xml files on disk are not required: the `DataSource.addDynamicDSGenerator()` API can be used to provide XML DataSource descriptors on the fly.

This API allows you to take advantage of additional sources of metadata to reduce hand-coding in DataSource descriptors. For example, you may have partial DataSource descriptors stored on disk, but use a DynamicDSGenerator to augment them with data derived from custom Java annotations or organization-specific naming conventions. This approach is complimentary with the built-in `autoDeriveSchema` system, since dynamically produced DataSource descriptors can still use `autoDeriveSchema`.

For more information on DataSource generation, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.serverds.DataSource.schemaBean

  com.smartgwt.client.docs.serverds.DataSource.autoDeriveSchema

  com.smartgwt.client.docs.serverds.DataSource.inheritsFrom

  com.isomorphic.datasource.DataSource.addDynamicDSGenerator()

Server Request Flow

When using the Smart GWT server framework, `DSRequests` go through the following flow:

1. DSRequest serialization: requests from DataSources are automatically serialized and delivered to the server.

2. DSRequest parsing: requests are automatically parsed by a servlet included with the Smart GWT server framework, and become `com.isomorphic.datasource.DSRequest` Java Objects.

3. Authentication, validation, and role-based security checks are performed based on declarations in your DataSource descriptor (`.ds.xml` file). For example, `requiresRole="manager"`.

4. DMI (Direct Method Invocation) and Server Scripts: custom logic can be run before or after the DataSource operation is performed, modifying the `DSRequest` or `DSResponse` objects, or can skip the DataSource operation and directly provide a `DSResponse`.

5. Persistence operation: the validated `DSRequest` is passed to a DataSource for execution of the persistence operation. The DataSource can be one of several built-in DataSource types (such as SQL or Hibernate) or a custom type.

6. The `DSResponse` is automatically serialized and delivered to the browser.

Most of these steps are entirely automatic—when you begin building a typical application using one of the built-in DataSource types, the *only* server-side source code files you will create are:

- `.ds.xml` files describing your business objects

- `.java` files with DMI logic expressing business rules

If you cannot use one of the built-in DataSource types (perhaps you have a pre-existing, custom ORM solution, or perhaps persistence involves contacting a remote server), you will also have Java code to implement persistence operations.

As your application grows, you can add Java logic or take over processing at any point in the standard server flow. For example, you can:

- replace the built-in servlet from step 2 (IDACall) with your own servlet, or place a servlet filter in front of it
- add your own Java validation logic
- subclass a built-in DataSource class and add additional logic before or after the persistence operation, such as logging all changes
- provide custom logic for determining whether a user is authenticated, or has a given role

For a more detailed overview of the server-side processing flow and documentation of available override points, see:

Smart GWT JavaDoc:

com.smartgwt.client.docs.ServerDataIntegration

Direct Method Invocation

DMI (Direct Method Invocation) allows you to declare what Java class and method should be invoked when specific `DSRequests` arrive at the server. A DMI is declared by adding a `<serverObject>` tag to your DataSource descriptor.

For example, the following declaration indicates that all DSRequests for this DataSource should go to the Java class `com.sample.DMIHandler`:

```
<DataSource ID="contacts" schemaBean="com.sample.Contact">
    <serverObject className="com.sample.DMIHandler"/>
</DataSource>
```

In this example, DMI will invoke a method on `com.sample.DMIHandler` named after the type of DataSource operation—`fetch()`, `add()`, `update()` or `remove()`.

The attribute `lookupStyle` controls how the server framework obtains an instance of `DMIHandler`. In the sample above, `lookupStyle` is not specified, so an instance of `DMIHandler` is created exactly as though the code `new DMIHandler()` were executed.

Other options for `lookupStyle` allow you to:

- target objects in the current servlet request or servlet session
- obtain objects via a factory pattern
- obtain objects via the Spring framework, including the ability to use Spring's "dependency injection" to set up the target object

As an alternative to `lookupStyle`, you can add small amounts of business logic directly to your .ds.xml file, avoiding the need for a separate .java file or formal class declaration. The section "Server Scripting" discusses this approach.

For more information on using `lookupStyle`, see:

Smart GWT JavaDoc:

com.smartgwt.client.docs.serverds.ServerObject

DMI Parameters

Methods invoked via DMI can simply declare arguments of certain types, and they are provided automatically.

For example, a common DMI method signature is:

```
public DSResponse fetch(DSRequest dsRequest) {
```

When this method is called via DMI, it will be passed the current `DSRequest`. If the method also needs the current `HttpServletRequest`, it can simply be declared as an additional parameter:

```
public DSResponse fetch(DSRequest dsRequest, HttpServletRequest request) {
```

This works for all of the common objects available to a servlet (such as `HttpSession`) as well as all Smart GWT objects involved in `DSRequest` processing (such as `DataSource`).

Parameter order is not important—available objects are matched to your method's declared parameters by type.

For more information on available DMI parameters, see:

Smart GWT JavaDoc:

com.smartgwt.client.docs.DMIOverview

Adding DMI Business Logic

A DMI *can* directly perform the required persistence operation and return a `DSResponse` populated with data, and in some use cases, this is the right approach.

However, if you are using one of the built-in DataSource types in Smart GWT, or you build a similar, re-usable DataSource of your own, DMI can be used to perform business logic that *modifies* the default behavior of DataSources.

Within a DMI, to invoke the default behavior of the DataSource and obtain the default `DSResponse`, call `dsRequest.execute()`. The following DMI method is equivalent to not declaring a DMI at all:

```
public DSResponse fetch(DSRequest dsRequest) throws Exception {
    return dsRequest.execute();
}
```

Given this starting point, we can see that it is possible to:

1. Modify the `DSRequest` before it is executed by the DataSource.

   For example, you might add criteria to a "fetch" request so that users who are not administrators cannot see records that are marked deleted.

   ```
   if (!servletRequest.isUserInRole("admin")) {
       dsRequest.setFieldValue("deleted", "false");
   }
   ```

2. Modify the `DSResponse` before it is returned to the browser.

   For example, adding calculated values derived from DataSource data, or trimming data that the user is not allowed to see. Typically, use `dsResponse.getRecords()` and iterate over the returned records, adding or modifying properties, then pass the modified List of records to `dsResponse.setData()`.

3. Substitute a completely different `DSResponse`, such as returning an error response if a security violation is detected

   To return a validation error:

   ```
   DSResponse dsResponse = new DSResponse();
   dsResponse.addError("fieldName", "errorMessage");
   return dsResponse;
   ```

   For this kind of error, the default client-side behavior will be to show the error in the UI component where saving was attempted.

   To return an unrecoverable error:

   ```
   DSResponse dsResponse =
       new DSResponse("Failure", DSResponse.STATUS_FAILURE);
   return dsResponse;
   ```

   For this kind of error, the default client-side behavior is a dialog box shown to the user, with the message "Failure" in this case. Customize this via the client-side API `RPCManager.setHandleErrorCallback()`.

4. Take related actions, such as sending an email notification.

   Arbitrary additional code can be executed before or after `dsRequest.execute()`, however, if the related action you need to perform is a persistence operation (such as adding a row to another SQL table), a powerful approach is to create *additional, unrelated* `DSRequests` that affect other DataSources, and `execute()` them.

   For example, you might create a DataSource with ID "changeLog" and add a record to it every time changes are made to other DataSources:

```
DSRequest extraRequest = new DSRequest("changeLog", "add");
extraRequest.setFieldValue("effectedEntity",
                           dsRequest.getDataSourceName());
extraRequest.setFieldValue("modifyingUser",
                           servletRequest.getRemoteUser());
// ... capture additional information ...
extraRequest.execute();
```

If you are using the Automatic Transaction management included in Power Edition, and you create a new `DSRequest` in a DMI, you **must** call `dsRequest.setRPCManager(rpcManager)` if you want the `DSRequest` to be included in the current transaction.

It often makes sense to create DataSources purely for server-side use—a quick idiom to make a DataSource inaccessible to browser requests is to add `requires="false"` to the `<DataSource>` tag—why this works is explained in the upcoming *Declarative Security* section.

Note that many of the DMI use cases described above can alternatively be achieved by adding simple declarations to your DataSource `.ds.xml` file—this is covered in more detail in the upcoming *Operation Bindings* section.

For more information on modifying the request and response objects, or executing additional requests, see:

Smart GWT Server JavaDoc:

com.isomorphic.datasource.DSRequest

com.isomorphic.datasource.DSResponse

For more information on error handling and display of errors, see:

Smart GWT JavaDoc:

com.smartgwt.client.rpc.RPCManager

com.smartgwt.client.widgets.form.DynamicForm

For a sample of DMI used to implement business logic, see:

Smart GWT Enterprise Showcase:

[smartgwtee/showcase/#sql_user_specific_data]

Returning Data

Whether you return data via DMI, via a custom DataSource, or via writing your own servlet and directly working with the `RPCManager` class, data that should be delivered to the browser is passed to the `dsResponse.setData()` API.

This API can accept a wide variety of common Java objects and automatically translate them into Record objects. For example, if you are responding to a fetch, the following Java will all translate to a List of Records if passed to `setData()`.

- Any Collection of Maps

  Each Map becomes a Record and each key/value pair in each Map becomes a Record attribute.

- Any Collection of Java Beans, that is, Java Objects that use the Java `get`*`PropertyName`*`()` / `set`*`PropertyName`*`()` naming convention

  Each Bean becomes a Record and each property on each bean becomes a Record attribute.

- Any Collection of DOM Elements (org.w3c.dom.Element)

  Each Element becomes a Record, and each attribute or subelement becomes a Record attribute.

Unlike typical XML, JSON, or RPC serialization systems, it is safe to directly pass persistent business objects to `dsResponse.setData()`. Most serialization systems, when given a persistent object such as a JPA or Hibernate Bean, will recursively serialize all connected objects. This frequently causes a multi-megabyte blob of data to be transmitted unless extra effort is expended to define a second, almost entirely redundant bean (called a DTO, or Data Transfer Object) where relevant data is copied before serialization.

In contrast, with Smart GWT, the list of fields in your DataSource is the full list of fields used by UI components, so it serves as a natural definition of what data to serialize, eliminating the need to define a redundant "DTO."

Serializing only data that matches field definitions is enabled by default for data returned from DMI, but can also be enabled or disabled automatically by setting `DataSource.dropExtraFields`.

Note that when delivering data to the browser, the Smart GWT server framework does not attempt to re-create Java Objects of the same type as the objects you supply to the server side `setData()` API. Systems such as GWT-RPC, which do attempt this, tend to require a huge amount of scaffolding code to be written in order to enable a comparatively trivial amount of functional code to be shared.

For the rare case of logic that can run unchanged on client and server, we recommend creating a small library of static methods and including it in both client and server-side projects.

For more information on how Java objects are translated to Records and how to customize the transformation, see:

- Smart GWT Server JavaDoc:

  com.isomorphic.js.JSTranslater.toJS()

Queuing & Transactions

Queuing is the ability to include more than one `DSRequest` in a single HTTP request.

When saving data, queuing allows multiple data update operations in a single HTTP request so that the operations can be performed as a transaction. When loading data, queuing allows you to combine multiple data loading operations into a single HTTP request without writing any special server-side logic to return a combined result.

Several UI components automatically use queuing. For example, the `ListGrid` supports an inline editing capability, including the ability to delay saving so that changes to multiple records are committed at once (`autoSaveEdits:false`). This mode automatically uses queuing, submitting all changes in a single HTTP request which may contain a mixture of "update" and "add" operations (for existing and new records respectively).

With respect to the steps described in the preceding section, *Server Request Flow*, when a request containing multiple `DSRequests` is received, several distinct `DSRequests` are parsed from the HTTP request received in step 1, steps 2-5 are executed for *each* `DSRequest`, and then all `DSResponses` are serialized in step 6.

This means that if any DataSource can support the "update" operation, the DataSource also supports batch editing of records in a ListGrid with no additional code, since this just involves executing the "update" operation multiple times. Likewise, in other instances in which components automatically use queuing (such as `removeSelectedData()` with multiple records selected, and multi-row drag and drop) implementing singular DataSource operations means that batch operations work automatically without additional effort.

If you use the `SQLDataSource` or `HibernateDataSource` with Power Edition or above, database transactions are used automatically, with a configurable policy setting (`RPCManager.setTransactionPolicy()`) as well as the ability to include or exclude specific requests from the transaction.

To implement transactions with your own persistence logic, make use of `dsRequest.getHttpServletRequest()`. Since this API will return the same `servletRequest` throughout the processing of a queue of operations, you can store whatever object represents the transaction—a `SQLConnection`, `HibernateSession`, or similar—as a `servletRequest` attribute.

For more information on transaction support, see:

- Smart GWT Server JavaDoc:

  com.isomorphic.rpc.RPCManager.setTransactionPolicy()

Queuing can be initiated manually by calling the client-side API `RPCManager.startQueue()`. Once a queue has been started, any user action or programmatic call that would normally have caused a `DSRequest` to be sent to the server instead places that request in a queue. Calling `RPCManager.sendQueue()` then sends all the queued `DSRequests` as a single HTTP request.

When the client receives the response for an entire queue, each response is processed in order, including any callbacks passed to DataBound Component methods.

A common pattern for loading all data required in a given screen is to start a queue, initiate a combination of manual data fetches (such as direct calls to `DataSource.fetchData()`) and automatic data fetches (allow a `ListGrid` with `setAutoFetchData(true)` to `draw()`), then finally call `sendQueue()`. Because in-order execution is guaranteed, you can use the callback for the final operation in the queue as a means of detecting that all operations have completed.

For more information on queuing, see:

- Smart GWT JavaDoc:

  com.smartgwt.rpc.RPCManager.startQueue()

Queuing, RESTHandler, and SOAs

The existence of *queuing* brings huge architectural benefits. In older web architectures, it was typical to define a unique object representing all the data that would need to be loaded for a particular screen or dialog, and a second object for any data that needed to be saved. This resulted in a lot of redundant code as each new screen introduced slightly different data requirements.

In contrast, queuing allows you to think of your code as a *set of reusable services* which can be combined arbitrarily to handle specific UI scenarios. New UI functionality no longer implies new server code—you will only need new server code when you introduce new fundamental operations, and, when you do introduce such operations, that is the only new code you'll need to write.

Using the `RESTHandler` servlet, this architecture can be extended to other, non-Smart GWT UI technologies that need the same services, as well as to automated systems. The `RESTHandler` servlet provides access to the same DataSource operations you use with Smart GWT UI components, with the same security constraints and server-side processing flow, but using simple XML or JSON over HTTP. The protocol used is the same as that documented for `RestDataSource`.

With the combination of queuing and the `RESTHandler` servlet, as you build your web application in the most efficient manner, you naturally create secure, reusable services that fit into the modern enterprise Service-Oriented Architecture (SOA).

For more information on the RESTHandler, see:

- Smart GWT Server JavaDoc:

  com.isomorphic.servlet.RESTHandler

Operation Bindings

Operation Bindings allow you to customize how `DSRequests` are executed with simple XML declarations.

Each Operation Binding customizes one of the four basic DataSource operations ("fetch", "add", "update," or "remove"). You specify which operation is customized via the `operationType` attribute.

Some basic examples:

- *Fixed criteria*: declare that a particular operation has certain criteria hardcoded. For example, in many systems, records are never actually removed and instead are simply marked as deleted or inactive. The following declaration would prevent users from seeing records that have been marked deleted—any value for the "deleted" field submitted by the client would be overwritten.

```
<DataSource ... >
    <operationBindings>
        <operationBinding operationType="fetch">
            <criteria fieldName="deleted" value="false"/>
        </operationBinding>
    </operationBindings>
</DataSource>
```

  Because this declaration affects the `DSRequest` before DMI is executed, it will work with any persistence approach, including custom solutions.

- *Per-operationType DMI*: declare separate DMIs for each operationType.

```
    <operationBinding operationType="fetch">
        <serverObject className="com.sample.DMIHandler"
                      methodName="doFetch"/>
    </operationBinding>
```

  This is important when using DMI to add business logic to a DataSource that already handles basic persistence operations, since most operations will not need DMIs, and it's simpler to write a DMI that handles one `operationType` only.

You can also use Operation Bindings to declare multiple *variations* of a DataSource operationType. For example, when doing a fetch, in one UI component you may want to specify criteria separately for each field, and in another UI component you may want to do a "full text search" across all the fields at once.

These are both operations of type "fetch" on the same DataSource, and they can be distinguished by adding an `operationId` to the Operation Binding. For example, if you had written a DMI method that performs full text search called "doFullTextSearch," you could declare an operationBinding like this:

```
<operationBinding operationType="fetch" operationId="fullTextSearch">
    <serverObject className="com.sample.DMIHandler"
                  methodName="doFullTextSearch"/>
</operationBinding>
```

You could now configure a `ListGrid` to use this Operation Binding via `grid.setFetchOperation("doFullTextSearch")`.

Another common use case for `operationId` is *output limiting*. Some DataSources have a very large number of fields, only some of which may be needed for a particular use case, like searching from a ComboBox. You can create a variation of the fetch operation that returns limited fields like so:

```
<operationBinding operationType="fetch" operationId="comboBoxSearch"
                  outputs="name,title"/>
```

Then configure a ComboBox to use this Operation Binding with `comboBox.setOptionOperationId("comboBoxSearch")`.

Setting `outputs` always limits the fields that are sent to the browser, regardless of the type of DataSource used. With the built-in DataSources, it also limits the fields requested from the underlying data store. Custom DataSources or DMIs that want to similarly optimize communication with the datastore can detect the requested outputs via `dsRequest.getOutputs()`.

For more information on features that can be configured via Operation Bindings, see:

- Smart GWT JavaDoc:

  com.smartgwt.client.docs.serverds.OperationBinding

Declarative Security

The Declarative Security system allows you to attach role-based access control to DataSource operations and DataSource fields, as well as create a mix of authenticated and non-authenticated operations for applications that support limited publicly-accessible functionality.

To attach role requirements to either a DataSource as a whole or to individual Operation Bindings, add a `requiresRole` attribute with a comma-separated list of roles that should have access.

Declarative Security is extremely powerful when combined with the ability to create variations on core operations via Operation Bindings. For example, if only users with the role “admin” should be able to see records marked as deleted:

```
<operationBinding operationType="fetch">
    <criteria fieldName="deleted" value="false"/>
</operationBinding>
<operationBinding operationType="fetch" operationId="adminSearch"
                  requiresRole="admin"/>
```

Declarative Security can also be used to control access to individual DataSource *fields*. Setting the `editRequiresRole` attribute on a DataSourceField will cause the field to appear as read-only whenever a user does not have any of the listed roles. Any attempts by such users to change the field value will be automatically rejected.

Similarly, the `viewRequiresRole` attribute will cause DataBound Components to avoiding showing the field at all, and values for the field will be automatically omitted from server responses. This behavior is automatic even if you build a custom DataSource or write DMI logic that returns data for the field, so it can be used regardless of how persistence is implemented.

The Declarative Security system can also be used to implement a mix of operations, some of which are publicly accessible while others may be accessed only by logged in users. To declare that a DataSource or Operation Binding may be accessed only by authenticated users, add `requiresAuthentication="true"`. You can also declare that individual fields are viewable or editable only by authenticated users, with the DataSourceField attributes `viewRequiresAuthentication` and `editRequiresAuthentication`.

For more information on using Declarative Security, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.serverds.OperationBinding.requiresRole

  com.smartgwt.client.docs.serverds.DataSource.requiresAuthentication

  com.smartgwt.client.docs.serverds.DataSource.viewRequiresRole

Declarative Security Setup

By default, the Declarative Security system uses the standard servlet API `httpServletRequest.getRemoteUser()` to determine whether a user is authenticated, and the API `httpServletRequest.isUserInRole()` to determine whether the user has a given role. In most J2EE security or JAAS security frameworks you might use, this API functions properly, and Declarative Security requires no setup steps – just start adding `requiresRole` attributes.

However, Declarative Security can be used with any security framework by simply calling `RPCManager.setAuthenticated(boolean)` to indicate whether the current request is from an authenticated user, and `RPCManager.setUserRoles()` to provide the list of roles. These APIs should be called before any requests are processed - this is typically done as a simple subclass of the built-in IDACall servlet.

Note further, although the terminology used is "roles," the Declarative Security system can also be used as a much finer-grained *capability security* system. Instead of using role names like "manager" in the `requiresRole` attribute, simply use capability names like "canEditAccounts" and use `RPCManager.setUserRoles()` to provide the current user's list of capabilities to the Declarative Security system.

For more information on setting up Declarative Security, see:

- Smart GWT Server Java Doc:

  com.isomorphic.rpc.RPCManager.setUserRoles()

  com.isomorphic.rpc.RPCManager.setAuthenticated()

  com.isomorphic.servlet IDACall

Non-Crud Operations

Some operations your application performs will not be "CRUD" operations - meaning they do not fall into the standard **C**reate, **R**etrieve, **U**pdate, **D**elete pattern (called "add", "fetch", "update" and "remove" in SmartClient). Smart GWT provides a few ways to execute such operations. The most convenient is simply to take an existing DataSource and declare a "custom" operation, like so:

```
<operationBinding operationType ="custom"
       operationId="customOperationId">
               ... settings ...
</operationBinding>
```

When you declare a custom operation, it means that the input and outputs of the operation are not constrained - they are not expected to conform to the DataSource fields, and will not be subject to basic integrity checks such as verifying that an "update" operation contains a value for the primary key field.

Although the custom operation being declared may not be strictly an operation on a specific DataSource, there is usually a DataSource that it is closely associated with, and declaring the operation in a DataSource file avoids the need to set up a separate mechanism. It also means that the custom operation can participate in queuing and transactions, and that all of the features of `operationBindings` can be used exactly as for other DataSource operations, including DMI and Declarative Security, as well as SQL Templating and Server Scripting (discussed in upcoming sections).

However, before declaring a custom operation, be sure you really have a non-CRUD operation. For example, if your operation returns a list of objects to be displayed in a grid, it's best represented as a "fetch" operation even if a SQL "SELECT" statement is not involved. Similarly, an operation that makes changes to DataSource Records should usually be declared with a CRUD `operationType`, otherwise, automatic cache synchronization won't work.

Specifically, all the following use cases should not use custom operations:

- adding logic before or after a CRUD operation - use DMI instead
- creating variations on CRUD operations - use `operationBinding.operationId` instead
- doing two or more CRUD operations in a single HTTP request – use Queuing instead

To invoke a custom operation, use `DataSource.performCustomOperation(`*`operationId`*`,`*`data`*`)`. The `data` parameter can contain any data (including nested structures) and is accessible server side via the `dsRequest.getValues()` API.

For more information on using Non-CRUD Operations, see:

- Smart GWT Java Doc:

com.smartgwt.client.performCustomOperation

Dynamic Expressions (Velocity)

In many places within the DataSource `.ds.xml` file, you can provide a *dynamic expression* to be evaluated on the server.

These expressions use the Velocity template language—a simple, easy-to-learn syntax that is used pervasively in the Java world.

Velocity works in terms of a *template context*—a set of objects that are available for use in expressions. Similar to DMI parameters, all Smart GWT and servlets-related objects are made available in the template context by default, including `dsRequest`, `ServletRequest`, `session` and so on.

References to objects in the template context have a prefix of '`$`', and dot notation is used to access any property for which a standard Java Bean "getter" method exists, or to access any value in a `java.util.Map` by its key. For example, `$httpSession.id` retrieves the current sessionId via `HttpSession.getId()`, and `$dsRequest.criteria.myFieldName` will retrieve a criteria value for the field "myFieldName" via `DSRequest.getCriteria()`, which returns a `Map`.

Some common use cases for dynamic expressions:

- **Server Custom Validators**

  The `serverCustom` validator type makes many common validation scenarios into single-line Velocity expressions:

```
<field name="shipDate" type="date">
   <validators>
      <validator
         type="serverCustom"
         serverCondition="$value.time > $record.orderDate.time"/>
   </validators>
</field>
```

- **Server-Assigned Criteria/Values**

  `<criteria>` and `<values>` tags allow you to modify the `DSRequest` before execution. For example, when implementing something like a "shopping cart," the following declaration would force all items added to the cart to be saved with the user's current servlet sessionId, and only allow the user to see his own items.

```
<operationBinding operationType="add">
   <values fieldName="sessionId" value="$session.id"/>
</operationBinding>
<operationBinding operationType="fetch">
   <criteria fieldName="sessionId" value="$session.id"/>
</operationBinding>
```

- **DMI Method Arguments**

  The `methodArguments` attribute can be added to an `<operationBinding>` to configure specific arguments that should be passed to a DMI method. For example, given a Java method:

```
List<Lead> getRelatedLeads(long accountId, boolean includeDeleted)
```

  You might call this method via a DMI declaration like:

```
<operationBinding operationType="fetch"
                  methodArguments="$criteria.accountId,false">
   <serverObject className="com.sample.DMIHandler"
                 methodName="getRelatedLeads"/>
</operationBinding>
```

  Because the `getRelatedLeads` method returns a List of Java Beans—a format compatible with `dsResponse.setData()`– there is no need to create or populate a `DSResponse`. Combining this with the `methodArguments` attribute allows you to call pre-existing Java business logic with *no Smart GWT-specific server code at all*, without even the need to import Smart GWT libraries code in your server-side logic.

- **Declarative Security** (`requires` attribute)

  Similar to `requiresRole` and `requiresAuthentication`, the `requires` attribute allows an arbitrary Velocity expression to restrict access control.

- **Mail Templates**

  By adding a `<mail>` tag to any `<operationBinding>`, you can cause an email to be sent if the operation completes successfully. A Velocity expression is allowed for each attribute that configures the email—`to`, `from`, `subject`, `cc`, and so on—as well as the message template itself. This makes it very easy to send out notifications when particular records are added or updated, or, with a "fetch" operation, send emails to a list of recipients retrieved by the fetch.

- **SQL/HQL Templating**

  When using SQLDataSource or HibernateDataSource in Power Edition and above, Velocity expressions can be used to customize generated SQL or replace it entirely. This is covered in its own section, *SQL Templating*.

If you have additional data or methods you want to make available for Velocity Expressions, you can add objects as attributes to the `servletRequest` - these are accessible via $servletRequest.getAttribute("attrName") (a shortcut of `requestAttributes.attrName` also works). You can alternatively add your own objects directly to the Velocity template context via `dsRequest.addToTemplateContext()`.

The Velocity template language can also call Java methods, create new variables, even execute conditional logic or iterate over collections. However, for any complex business logic, consider using Server Scripting instead (described in the next section).

For more information on Velocity-based Dynamic Expressions:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.serverds.VelocitySupport

  com.smartgwt.client.docs.serverds.Validator.serverCondition

- Smart GWT Server Java Doc:

  com.isomorphic.datasource.DSRequest.addToTemplateContext()

- Velocity User Guide (from the Apache foundation)

  velocity.apache.org/user-guide

Server Scripting

Smart GWT allows you to embed "scriptlets" directly in your .ds.xml file to take care of simple business logic without having to create a separate file or class to hold the logic.

These scriptlets can be written in any language supported by the Java" JSR 223" standard such as Groovy, JavaScript, Velocity, Python, Ruby, Scala and Clojure.

The two primary use cases for server scripts are:

1. DMI scriptlets: these scriptlets are declared by adding a `<script>` tag to an `<operationBinding>` or `<DataSource>` tag. Like DMI logic declared via `<serverObject>`, DMI scriptlets can be used to add business logic by modifying the `DSRequest` before it is executed, modifying the default `DSResponse`, or taking other, unrelated actions.

2. scriptlet validators: these scriptlets are declared by adding a `<serverCondition>` tag to a `<validator>` definition. Like a validator declared via `<serverObject>`, a scriptlet validator defines whether data is valid by running arbitrary logic, then returning true or false.

For example, the following scriptlet enforces a security constraint where all operations on the DataSource will involve the `sessionId`, so a user can only view and modify their own records.

```
<DataSource...>
    <script language="groovy">
        String sessionId = session.getId();

        if (DataSource.isAdd(dsRequest.getOperationType())) {
           dsRequest.setFieldValue("sessionId", sessionId);
        } else {
           dsRequest.setCriteriaValue("sessionId", sessionId);
        }

        return dsRequest.execute();
     </script>
  ...
```

There is no need for a formal class or method definition - the context of a DMI Script is always the same, and the Server Scripting system avoids the need to add this "boilerplate code".

Using scriptlets has a couple of major advantages as compared to using `<serverObject>`:

1. Simplicity & Clarity: scriptlets put business logic right next to the relevant persistence operation instead of requiring that you look in a separate .java file

2. Faster Development Cycle: scriptlets are compiled and executed dynamically, so you do not need to recompile or redeploy your server code to try out changes to scriptlets. Just edit your DataSource, and then either reload the page or retry the operation. The SmartClient Server framework automatically notices the changed DataSource and uses the updated scriptlet.

Note that scriptlets are only recompiled when you change them, so will only be compiled once ever in the final deployment of your application.

The ability to use Groovy as a "scripting language" is particularly powerful:

1. Developers do not have to know more than one language to work with the code for your application (Groovy syntax is, for all intents and purposes identical to Java syntax).

2. Scriptlets can easily be moved into normal .groovy or .java files if they are identified as reusable, become too large to manage in a .ds.xml file, or if the (small) performance boost of compiling them to .class files is desired. There is no need to translate from some other language into Java
For these reasons we recommend use of Server Scripting with the Groovy language even for teams that would not normally consider adopting a "scripting language".

For an example of Server Scripting see:

- Smart GWT Enterprise Showcase:

  http://www.smartclient.com/smartgwtee/showcase/#_Featured.Samples_Server.Examples_Server.Scripting

For more information on Server Scripting

- Smart GWT Server Java Doc:

  com.smartgwt.client.docs.ServerScript

Including Values from Other DataSources

Frequently, you will need to show a UI that includes fields from two related DataSources - something typically accomplished in SQL with a "join". For simple cases of this, you can use `DataSourceField.includeFrom`.

For example, a DataSource `stockItem` may store information about items for sale in a store, including "itemName" and "price". A related DataSource `orderItem` may store the "id" and "quantity" of a `stockItem` that was ordered. When the user views all the `orderItem`s in a order, they want to see the "itemName"s of the related `stockItem`s, not their "id"s.

To accomplish this, you can declare an additional field in the `orderItem` DataSource like so:

```
<field includeFrom="stockItem.itemName"/>
```

Now, when the `orderItem` DataSource responds to a "fetch" request, it will include an additional field "itemName" which comes from the related `stockItem` DataSource. Note how the field declared in XML above is not given a "name" attribute - the name is optional in this case, and will default to the name of the included field.

In order for included fields to work, the `foreignKey` attribute must be used to declare the relationship between the two DataSources. In this case, there might be a field `orderItem.stockItemId` with `foreignKey="stockItem.id"`). Once relationships are declared, multiple fields may be included from multiple different DataSources by simply adding more `includeFrom` declarations.

When `includeFrom` is used with the built-in SQLDataSource, HibernateDataSource or JPADataSource (when the provider is Hibernate), an efficient SQL join is used to include the field from the related DataSource, and search criteria and sort directions work normally with included fields.

For other kinds of DataSources, `includeFrom` operates by first fetching records from the main DataSource, then fetching related records from the included DataSource. In this case search criteria and sort directions specified for included fields only work if data paging is not in use.

In the upcoming discussion of SQL Templating we'll see how to do more advanced joins as well as make use of SQL features such as expressions, grouping and aggregation.

SQL Templating

A DataSource declared with `serverType="sql"` uses the `SQLDataSource`, which automatically generates and executes SQL statements against a database in response to `DSRequests` sent by the client.

When using the `SQLDataSource` with the Power Edition of Smart GWT, *SQL Templating* enables fine-grained customization of generated SQL.

The SQL generator in Power Edition can take the `DSRequests` generated by DataBound components and automatically handle:

- Generation of a where clause from complex criteria, including nested “and” and “or” sub-expressions
- Database-specific SQL for the most efficient ranged selections on each platform, for fast data paging
- Multi-level sorting including support for sorting by displayed rather than stored values
- Several different styles of storing basic types like booleans and dates, for adapting to existing tables

When you inevitably have to customize the generated SQL for a particular use case, it’s critical to preserve as much of this powerful, automatic behavior as possible.

Most systems that allow customization of generated SQL provide only an all-or-nothing option: if you need to customize, you write the complete SQL query from scratch, and handle all database-specific SQL yourself.

In contrast, the SQL Templating system lets you change small parts of the generated SQL while leaving all the difficult, database-specific SQL up to Smart GWT. SQL Templating also allows you to take advantage of database-specific features where appropriate, without losing automatic SQL generation for standard features.

The following table summarizes the SQL statements that are generated and how the `DSRequest` is used (note, these aren't the actual statements – additional SQL exists to handle data paging and database-specific quirks):

Type	SQL statement	DSRequest usage
`fetch`	SELECT <selectClause> FROM <tableClause> WHERE <whereClause> GROUP BY <groupClause> ORDER BY <orderClause>	`data` becomes <whereClause> `sortBy` becomes <orderClause> `outputs` becomes <selectClause>
`add`	INSERT INTO <tableClause> <valuesClause>	`data` becomes <valuesClause>
`update`	UPDATE <tableClause> SET <valuesClause> WHERE <whereClause>	`data` becomes <valuesClause> and <whereClause> (primary key only)
`remove`	DELETE FROM <tableClause> WHERE <whereClause>	`data` becomes <whereClause> clause (primary key only)

To customize SQL at a per-clause level, you can add tags to your `<operationBinding>` named after SQL clauses. Each clause allows a Velocity template, and the default SQL that would have been generated is available to you as a Velocity variable:

XML Tag	Velocity Variable	SQL Meaning
`<selectClause>`	$defaultSelectClause	List of columns or expressions appearing after `SELECT`
`<tableClause>`	$defaultTableClause	List of tables or table expressions appearing after `FROM`
`<whereClause>`	$defaultWhereClause	Selection criteria appearing after `WHERE`
`<valuesClause>`	$defaultValuesClause	List of expressions appearing after `SET` (for `UPDATE`) or list of column names and `VALUES()` around list of expressions (for `INSERT`)
`<orderClause>`	$defaultOrderClause	List of columns or expressions appearing after `ORDER BY`
`<groupClause>`	<none>	List of columns or expressions appearing after `GROUP BY`

As a simple example, in an order management system, you may want to present a view of all orders for items that are not in stock. Given two tables, `orderItem` and `stockItem`, linked by id, you could add an `<operationBinding>` to the DataSource for the `orderItem` table:

```
<operationBinding operationType="fetch" operationId="outOfStock">
    <tableClause>orderItem, stockItem</tableClause>
    <whereClause>orderItem.stockItem_id == stockItem.id AND
                 stockItem.inStock == 'F' AND ($defaultWhereClause)
    </whereClause>
</operationBinding>
```

Note the use of `$defaultWhereClause`—this ensures that any criteria submitted to this operation still work. Data paging and sorting likewise continue to work.

It is also possible to override the entire SQL statement by using the `<customSQL>` tag. This makes it very easy to call stored procedures:

```
< operationBinding operationType ="remove">
    <customSQL> call deleteOrder($criteria.orderNo)</customSQL>
</operationBinding>
```

When customizing a "fetch" operation, use clause-by-clause overrides instead where possible. Using the `<customSQL>` tag for a "fetch" operation disables the use of efficient data paging approaches that can only be used when Smart GWT knows the general structure of the SQL query. However, if you know that your customized SQL is still compatible with the SQL added for data paging, you can use the `operationBinding.sqlPaging` attribute to re-enable it.

For more information on SQL Templating, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.CustomQuerying

SQL Templating — Adding Fields

A customized query can return additional fields that aren't part of the DataSource's primary table, and even allow criteria to be automatically applied to such fields.

For the common case of incorporating a field from another table, declare a field as usual with a `<field>` tag, then add the attribute `tableName="otherTable"`. Setting `tableName` enables a field to be fetched from another table and used in the `WHERE` clause, but automatically excludes the field from the SQL for any `operationType` except "fetch."

For example, given the `orderItem` and `stockItem` tables from the preceding example, imagine `stockItem` had a column `itemName` that you want to include in results from the `orderItem` DataSource.

```
<DataSource ID="orderItem" serverType="sql" tableName="orderItem"
            autoDeriveSchema="true">
    <fields>
        <field name="itemName" type="text" tableName="stockItem"/>
    </fields>
    <operationBindings>
       <operationBinding operationType="fetch">
           <tableClause>orderItem, stockItem</tableClause>
           <whereClause>orderItem.stockItem_id == stockItem.id AND
                   ($defaultWhereClause)
           </whereClause>
        </operationBinding>
    </operationBindings>
</DataSource>
```

This approach can be extended to any number of fields from other tables.

For an example of SQL Templating being used to add a searchable field, see:

- Smart GWT Enterprise Showcase

smartgwtee/showcase/#large_valuemap_sql

In some cases, you may have several different Operation Bindings that use different sets of added fields. In this case, you can set `customSQL="true"` on the `<field>` element to turn off automatic generation. Then, use the following `<operationBinding>` properties to control whether SQL is generated for the field on a per-`<operationBinding>` basis.

Setting	Meaning
`customValueFields`	Comma-separated list of fields to allow in `SELECT` clause despite being `customSQL="true"`
`customCriteriaFields`	Comma-separated list of fields to allow in `WHERE` clause despite being `customSQL="true"`
`excludeCriteriaFields`	Comma-separated list of fields to **exclude** from `$defaultWhereClause`

You can also define custom SQL on a per-field basis rather than a per-clause basis using the following properties on a `<field>`:

Setting	Meaning
`customSelectExpression`	Expression to use in `SELECT` and `WHERE` clauses
`customUpdateExpression`	Expression to use in `SET` clause of `UPDATE`
`customInsertExpression`	Expression to use in `VALUES` clause of `INSERT`. Defaults to `customUpdateExpression`

`customSelectExpression` alone is enough to create a searchable field that uses a SQL expression to derive its value, which can be used for SQL-based formatting, including combining values from multiple database columns into one logical DataSource field. For example, the following field definition would combine `firstName` and `lastName` columns at the database:

```
<field name="fullName"
       customSelectExpression="CONCAT(CONCAT(firstName, ' '), lastName)"
/>
```

Applied in combination, the `custom..Expression` properties can be used to create a field that uses SQL expressions to map between a stored SQL value and the value you want to use in Smart GWT UI components. This can be used to handle legacy formats for date values, database-specific variations of boolean storage including “bit vector” columns, and other use cases. For example, you might store a price in cents, but want to work in the UI in terms of dollars:

```
<field name="unitPrice" type="float"
       customSelectExpression="unitPrice / 100"
       customUpdateExpresion="$values.unitPrice * 100" />
```

Before using these properties, take a look at `DataSourceField.sqlStorageStrategy`, which encapsulates some common scenarios as a single setting.

For more information on SQL Templating, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.serverds.DataSourceField.customSQL

  com.smartgwt.client.docs.serverds.OperationBinding.customCriteriaFields

  com.smartgwt.client.docs.serverds.DataSourceField.customSelectExpression

  com.smartgwt.client.docs.serverds.DataSourceField.sqlStorageStrategy

For a sample of SQL Templating involving a complex, aggregated query that still supports paging and search, see:

- Smart GWT Enterprise Showcase

  smartgwtee/showcase/#sql_dynamic_reporting

Why focus on `.ds.xml` files?

Having read about operation bindings, declarative security, dynamic expressions and SQL Templating, you probably now realize that 95% of common web application use cases can be handled with simple settings in a `.ds.xml` file. This short section is a reminder of why this brings tremendous benefits.

- Declarative

  Far more compact than creating a Java class to hold equivalent logic, and can be read and understood by people who would not be able to read equivalent Java code, such as QA engineers, UI engineers or product managers with XML and SQL skills.

- Centralized

  Security rules and other business rules appear right in the business object definition, where they are more easily found.

- Secure

  `.ds.xml` files are evaluated server-side, so all business rules declared there are securely enforced. By driving client-side behavior from secure server declarations, you avoid the common error of implementing a business rule client-side only, and forgetting to add server enforcement.

  Further, the DataSource definition delivered to the client automatically omits all declarations that only drive server-side behaviors (such as DMI), so there is no information leakage.

  Finally, in sensitive contexts like SQL Templating, automatic quoting is applied, making it far more difficult to accidentally create common security flaws like SQL injection attacks.

- Faster development cycle

  To test a new functionality in a DataSource `.ds.xml` file, just reload the web page—the Smart GWT server framework automatically notices the modified DataSource. No compilation and deployment step required.

Custom DataSources

You can create a DataSource that calls existing business logic by simply using DMI to declare what Java method to call for each operation. This is a good approach if you have only a few DataSources, or while you are still learning the basics.

However, Smart GWT allows you to create a custom, reusable DataSource classes in Java, which can then be used with an unlimited number of `.ds.xml` files. Do this when:

- you have several DataSources that all use a similar persistence approach, and DMI declarations and associated code would be highly repetitive

- you are using a built-in DataSource such as SQLDataSource, but you would like to extend it with additional behaviors

In both cases, you use the `serverConstructor` attribute of the `<DataSource>` tag to indicate the Java class you would like to use. Your Java class should extend the DataSource class that you are using for persistence, or, if writing your own persistence code, extend `com.isomorphic.datasource.BasicDataSource`.

Providing responses from a custom DataSource works similarly to DMI—there are 4 methods on a DataSource, one per DataSource operation type, each of which receives a `DSRequest` and returns a `DSResponse`. They are `executeFetch`, `executeUpdate`, `executeAdd` and `executeRemove`.

If you are extending a built-in DataSource that provides persistence, you can override one or more of these methods, add your custom logic, and call the superclass implementation with the Java `super` keyword.

If you are implementing your own persistence, you need to provide an implementation for each of the operations you plan to use. Once these methods are implemented, convenience methods such as `DataSource.fetchById()` become functional automatically. Use `getFieldNames()`, `getField()` and the APIs on the `DSField` class to discover the field definitions declared in the `.ds.xml` file. You can return data in the `DSResponse` in exactly the same formats as are allowed for DMI.

A fifth override point, `DataSource.execute()`, can be used for common logic that should apply to all four DataSource operations. The `execute()` method is called before operation-specific methods such as `executeFetch()` and is responsible for invoking these methods. Here again, use `super` to allow normal execution of operation types you don't wish to centrally customize.

You can also add custom attributes to your DataSource `.ds.xml` file. The APIs `DataSource.getProperty()` and `DSField.getProperty()` allow you to detect added attributes at the DataSource and DataSourceField level respectively. Use these attributes to configure your persistence behavior (for example, the URL of a remove service to contact) or use them to control additional features you add to the built-in persistent DataSources.

For more information on creating custom DataSources, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.WriteCustomDataSource

9. Extending Smart GWT

Smart GWT provides a rich set of components and services to accelerate your development, but from time to time, you may want to extend outside the box of prefabricated features. For example, you might need a new user interface control, or special styling of an existing control, or a customized data-flow interaction. With this in mind, we have worked hard to make Smart GWT as *open* and *extensible* as possible.

An earlier chapter, *Smart GWT Server Framework*, outlined the approaches to extending Smart GWT on the *server*. This chapter outlines the customizations and extensions that you can make on the *client*.

New Components

If you need to extend beyond the customizable properties of the standard Smart GWT component set, you can create entirely new components.

New components are usually based on one of the following foundation classes: `Canvas`, `StatefulCanvas`, `Layout`, `HLayout`, `VLayout`, `HStack`, or `VStack`.

The three most common approaches to build a new Smart GWT visual component are:

1. Create a subclass of any built-in layout class that generates and manages a set of other components.

   Many of Smart GWT built-in components are built in this fashion, for example, the `Window` class is a subclass of `Layout` that automatically creates header, footer and body components.

   Use layout subclasses to build high-level compound components and user interface patterns. For example, you could define a new class that combines a summary grid, toolbar, and detail area into a single reusable module.

2. Create a `Canvas` subclass that generates and configures a set of other foundation components.

   Use this approach if your custom component does not resemble any of the built-in layout classes (or any combination of nested layouts). The `Slider` control included with Smart GWT is a good example of this pattern: a `Slider` is created out of a `Canvas` parent, `StretchImg` track element, and draggable `Img` thumb element. By reusing foundation components, you avoid browser inconsistencies in HTML and CSS rendering, event handling, and other areas.

3. Create a `Canvas` subclass that contains your own HTML and CSS template code.

   This approach provides the most flexibility to create components using any feature of HTML and CSS and is also the best approach for embedding third-party JavaScript widgets within Smart GWT containers. However, it also requires that you test, optimize, and maintain your code on all supported web browsers. Whenever possible, you should use Smart GWT foundation components instead to avoid browser inconsistencies.

For more information on creating components from raw HTML and CSS and integrating third-party JavaScript components, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.DomIntegration

Before you begin development of an entirely new component, try the Smart GWT Developer Forums at *forums.smartclient.com*. Other developers may have created similar components, or Isomorphic Software may have already scheduled, specified, or even implemented the functionality you need.

You can also contact Isomorphic regarding Feature Sponsorship to have the component added to the product along with documentation and running samples, so you won't need to build and maintain the code yourself. For Feature Sponsorship, contact Isomorphic at *services@isomorphic.com*.

New Form Controls

New form controls are frequently implemented by taking a built-in form control and adding an icon that opens a custom value picker.

To create a new form control with this approach:

1. Create a subclass of `TextItem` or `StaticTextItem`.

2. Add a picker icon to instances of your control (see `FormItem.addIcon()`).

3. Build a custom picker based on any standard or custom Smart GWT components and services (see above). The `Window` component is a common choice.

4. Respond to end-user click events on that icon to show your picker (see `FormItemIcon.addFormItemClickHander()`) to show your picker.

5. Update the value of the form control based on user interaction with the picker (see `FormItem.setValue()`).

6. Hide the picker when appropriate.

Custom pickers are often implemented in SmartClient `Window` or `Dialog` components.

New form controls can alternatively be implemented via the `CanvasItem` class, which allows any Smart GWT component to be embedded into a `DynamicForm` in order to display and edit a field value. For example, a `CanvasItem` can be used to embed a `ListGrid`, which might be used to provide an alternative to HTML's multiple select input, displaying extra styling or additional controls (such as a "Select All" button). Or, a `CanvasItem` could contain a second `DynamicForm` where multiple `FormItems` are used to edit a single field value. A `CanvasItem` could even provide a complete interface for editing a nested Record.

However, when using `CanvasItem`, remember that you may also want to support inline editing within a `ListGrid`. A custom form control based on a pop-up picker dialog works well with inline editing because the control remains the same height as most of the built-in form controls. While a `CanvasItem` can be used to embed an arbitrarily complex interface into a form, the approach of pop-up picker often makes more sense if the custom form control will also be used for inline editing in grids.

For more information on new form controls, see:

- Smart GWT Java Doc:

  com.smartgwt.client.widgets.form.fields.CanvasItem

- Smart GWT Enterprise Showcase

  http://www.smartclient.com/smartgwt/showcase/#nested_editor

Switching Theme

Themes allow you to change the overall look-and-feel of your Smart GWT user interface. You can “re-skin” an application to match corporate branding, to adhere to usability guidelines, or even to personalize look & feel to individual user preferences.

Smart GWT includes several default themes. To get a visual preview of the default themes, use the online Showcase, which includes a control to dynamically switch themes.

In your own project, you switch to a different theme by inheriting a GWT module in your `.gwt.xml` file. We recommend using the Enterprise, Enterprise Blue or Graphite themes, as these offer the best performance and offer automatic adaptation for tablet and phone devices. These skins can be found in the client-side .jar file for your edition:

Enterprise or Evaluation: smartgwtee.jar

Power: smartgwtpower.jar

Pro: smartgwtpro.jar

LGPL: smartgwt.jar

Several older themes are still available, and can be found in smartgwt-skins.jar (regardless of your edition).

In all cases, the resources for the themes are found under the Java package name com.smartclient.theme. An IDE such as Eclipse will allow you to browse the .jar files to find available themes. To switch themes, add an <inherits> tag with the fully qualified Java package name of the .gwt.xml file for the theme, after other Smart GWT inherits

```
<inherits
name="com.smartclient.theme.graphite.Graphite"/>
```

Note: Smart GWT Pro/EE starter projects include `.gwt.xml` files that inherit a single GWT module in order to include all Smart GWT components in the project. For example, in the evaluation SDK, you'll see:

```
<inherits name="com.smartgwtee.SmartGwtEE"/>
```

This `<inherits>` tag implicitly includes a theme for the Smart GWT components. Before adding a new `<inherits>` tag for a different theme, add `NoTheme` to the end of the `name` attribute, like so:

```
<inherits name="com.smartgwtee.SmartGwtEENoTheme"/>
```

This revised `<inherits>` tag includes just the Smart GWT components, with no theme.

If you forget to add `NoTheme`, you will be **loading two themes** and the result will be a strange combination of both themes. If you see strange visual anomalies or theme-related files being downloaded from two different `images` directories, check your `<inherits>` tags.

Customizing Themes

In the previous section we looked at how to find and browse the default themes. In each theme you will find a `public` folder with 3 sets of resources:

Resource	Contains
`skin_styles.css`	a collection of CSS styles that are applied to parts of visual components in various states (e.g. `cellSelectedOver` for a selected cell in a grid with mouse-over highlighting)
`images`	a collection of small images that are used as parts of visual components when CSS styling is not sufficient (e.g. `TreeGrid/folder_closed.gif`)
`load_skin.js`	high-level programmatic styling (e.g. `listGrid.alternateRecordStyles`) and sizes for fixed-size elements (e.g. `window.edgeSize`)

The best way to create a custom skin is to copy an existing skin that most closely matches your intended look and feel and modify it. For example,

let's say you wanted to customize the built-in "Enterprise" skin and call the resulting skin "BrushedMetal." The procedure is as follows:

7. Use any .zip-compatible tool to unzip the .jar file and copy the entire contents of the "Enterprise" skin into a new folder called "BrushedMetal."

8. Edit the load_skin.js file. Find the line near the top of the file that reads:

   ```
   isc.Page.setSkinDir("[ISOMORPHIC]/skins/Enterprise/")
   ```

   and change it to:

   ```
   isc.Page.setSkinDir("[ISOMORPHIC]/skins/BrushedMetal/")
   ```

9. Rename Enterprise.gwt.xml to BrushedMetal.gwt.xml and change the path sc/skins/Enterprise to sc/skins/BrushedMetal within this file.

10. Now you're ready to customize the new skin. You can do so by modifying any of the files listed in the preceding table inside your new skin directory. When modifying your custom skin, best practice is to group all changes in skin_styles.css and load_skin.js near the end of the file, so that you can easily apply your customizations to future, improved versions of the original skin.

11. Create a .jar for the skin. Eliminate the outer directory structure com/smartclient/theme and replace with com/*mycompany*/theme. Use any .zip-compatible tool to create the .jar file, and add it to your Smart GWT project.

12. Switch to your new skin by changing the <inherits> tags in your .gwt.xml file, as covered in the previous section.

Packaging your new skin as a GWT module is convenient for sharing it across projects, but not actually required. Another approach is to take the unzipped skin files and place them under the war directory of your project, then add a <script src=> tag to your .html bootstrap file to load load_skin.js. In this case the setSkinDir() call from step #2 should use the relative path to the unzipped files – for example, if you placed the skin in war/skins/BrushedMetal then the call should be isc.Page.setSkinDir("skins/BrushedMetal"). This is a good approach to use while making a series of changes to the skin.

For more information on skinning, see:

- Smart GWT Java Doc:

  com.smartgwt.client.docs.Skinning

10. Tips

Beginner Tips

Use the Developer Console.

The Developer Console contains several extremely useful diagnostic and runtime inspection tools and is where Smart GWT logs errors and warnings. You should always have the Developer Console open while developing with Smart GWT.

Architecture Tips

Don’t mix Smart GWT and plain GWT components unless forced to.

Wherever possible, build your UI entirely out of Smart GWT widgets rather than using a mixture of plain GWT (com.google.gwt) widgets and Smart GWT widgets.

This is required because there are no interoperability standards that allow two Ajax component systems (including core GWT widgets) to coordinate on management of tab order and keyboard focus, layering and modality, pixel-perfect layout, and accessibility.

Smart GWT does have limited interoperability support that allows a Smart GWT widget to be added to a GWT container and allows a GWT widget to be added to a Smart GWT container, and it’s appropriate to use this for:

- incremental migration to Smart GWT, such as introducing singular, sophisticated Smart GWT components like the `Calendar` or `CubeGrid` to an existing GWT application

- using sophisticated third-party GWT widgets within Smart GWT, where Smart GWT doesn't have corresponding built-in functionality

However, you should **never** place GWT widgets within a Smart GWT container that is in turn within a GWT container. Until interoperability standards emerge, intermixing widgets in this way is considered unsupported usage, and any issue reports resulting from such usage will not be considered bugs.

Defer creation and drawing of components until they are shown to the end user.

Creating and drawing all of your components on the `onModuleLoad()` function will lead to poor start time for larger applications. Instead, create and draw only the components required for the initial view.

Defer creation of components by waiting until the user navigates to the view. For example, to create the components which appear in a tab only when the user selects the tab, use the `TabSelected` event in conjunction with `Tab.setPane()`.

To reclaim all memory for a component that you no longer need, call `destroy()` on it, then allow it to be garbage collected by removing all references to it as usual. Note that `destroy()` is permanent, and once you have called `destroy()` on a component, calling any other API is expected to fail. Destroying a parent component automatically destroys all children.

To reclaim *some* memory from a component that you wish to reuse later, call `clear()`. This removes all HTML rendered by the component and its children. You can call `draw()` to recreate the component's HTML later.

Multiple Primary Keys

DataSources with multiple primary keys are supported by most components and interactions that involve primary keys, including ListGrid editing as well as the server-side SQL, Hibernate and JPA connectors. Multiple primary key are not supported for defining tree structures, or for certain convenience features like `dataSourceField.includeFrom`.

However, if you have a choice, it's preferred to use a singular primary key of type "sequence". Validation logic and/or a unique constraint in the data store can be used to ensure the uniqueness of values across multiple fields.

If you are stuck with a data model involving multiple primary keys and you need to use a feature that doesn't support multiple primary keys, you can use a "synthetic" primary key: declare a single primary key in your DataSource, then generate values for this field by combining the values of the primary key fields in the underlying data store.

HTML and CSS Tips

Use Smart GWT components and layouts instead of HTML and CSS, whenever possible.

The goal is to avoid browser-specific HTML and CSS code. The implementations of HTML and CSS vary widely across modern web browsers, even across different versions of the same browser. Smart GWT components buffer your code from these changes, so you do not need to test continuously on all supported browsers.

Avoid FRAME and IFRAME elements whenever possible.

Frames essentially embed another instance of the web browser inside the current web page. That instance behaves more like an independent browser window than an integrated page component. Smart GWT's dynamic components and background communication system allow you to perform fully integrated partial-page updates, eliminating the need for frames in most cases. If you must use frames, you should explicitly clear them with `frame.document.write("")` when the parent page is unloaded, to avoid memory leaks in Internet Explorer.

Manipulate Smart GWT components only through their published APIs.

Smart GWT uses HTML and CSS elements as the “pixels” for rendering a complex user interface in the browser. It is technically possible to access these elements directly from the browser DOM (Document Object Model). However, these structures vary by browser type, version, and mode, and they are regularly improved and optimized in new releases of Smart GWT. The only stable, supported way to manipulate a Smart GWT component is through its published interfaces.

Set your browser to HTML5 mode.

Internet Explorer 9 and onward are crippled if HTML5 mode is disabled, therefore, you must use the HTML5 DOCTYPE <!DOCTYPE html> with these browsers.

Unfortunately, Internet Explorer 8 has poorer performance and some minor, uncorrectable cosmetic defects when used with the HTML5 DOCTYPE. If possible, use the HTML5 DOCTYPE with Internet Explorer 9 and above, and omit the DOCTYPE with Internet Explorer 8 and below. If this is not possible, just use the HTML5 DOCTYPE for all versions of Internet Explorer.

The HTML5 DOCTYPE is also recommended for all other supported browsers.

11. Evaluating Smart GWT

This chapter offers advice for the most effective approaches to use when evaluating Smart GWT for use in your project or product.

Which Edition to Evaluate

Smart GWT comes in several editions, including a free edition under the Lesser GNU Public License (LGPL).

We always recommend using the commercial edition for evaluation. The reason is simply that applications built on the commercial edition can be easily converted to the LGPL version without wasted effort, but the reverse is not true.

For example, the commercial edition of Smart GWT includes a sample project with a pre-configured Hypersonic SQL Database, which you can use to evaluate all of the capabilities of Smart GWT's UI components without ever writing a line of server code, using simple visual tools to create and modify SQL tables as needed.

If you ultimately decide *not* to purchase a commercial license, Smart GWT's DataSource architecture allows for plug-replacement of DataSources without affecting any UI code or client-side business logic. So, you can simply replace the SQL DataSources you used during evaluation with an alternative implementation, and there is no wasted work.

Similarly, if part of your evaluation involves connecting to pre-existing Java business logic, Smart GWT Direct Method Invocation (DMI) allows you to route DataSource requests to Java methods by simply declaring the target Java class and method in an XML file. To later migrate to Smart GWT LGPL, just replace your DMI declarations with your own system for serializing and de-serializing requests and routing them to Java methods.

If you wrote any server-side pre- or post-processing logic to adapt Smart GWT's requests and responses to your business logic methods, this will continue to be usable if you decide to write and maintain a replacement for Smart GWT DMI. No code is thrown away and none of your UI code needs to change.

In contrast, if you were to evaluate using the LGPL edition and implement REST-based integration, upon purchasing a license you will immediately want to switch to the more powerful, pre-built server integration instead, which also provides access to all server-based features. In this scenario you will have wasted time building a REST connector during evaluation *and* given yourself a false perception of the learning curve and effort involved in using Smart GWT.

Evaluating the commercial edition gives you a more effective, more accurate evaluation process and avoids wasted effort.

Evaluating Performance

Smart GWT is the highest performance platform available for web applications, and you can easily confirm this during your evaluation.

However, be careful to **measure correctly**: much of the performance advice you may encounter applies to web *sites*, is focused on reducing initial load time, and can actually drastically reduce responsiveness and scalability if applied to a web *application*.

Unlike many web sites, web *applications* are visited repeatedly by the same users on a frequent basis, and users will spend significant time actually using the application.

To correctly assess the performance of a web *application*, what should be measured is performance when completing a typical series of tasks.

For example, in many different types of applications a user will search for a specific record, view the details of that record, modify that record or related data, and repeat this pattern many times within a given session.

To assess performance in this scenario, what should be measured is requests for *dynamically generated responses* - for example, results from a database query. Requests for static files, such as images and CSS style sheets, can be ignored since these resources are cacheable—these requests will not recur as the user runs through the task multiple times, and will not recur the next time the user visits the application.

Focusing on dynamic responses allows you to measure:

- *responsiveness*: typically a dynamic response means the user is blocked, waiting for the application to load data. It's key to measure and minimize these responses because these are the responses users are actually waiting for in real usage.

- *scalability:* dynamic responses represent trips to a data store and processing by the application server—unlike requests for cacheable resources, which occur only once ever per user, dynamically generated responses dictate how many concurrent users the application can support.

Using network monitoring tools such as Firebug (*getfirebug.com*) or Fiddler (*fiddlertool.com*), you can monitor the number of requests for *dynamic data* involved in completing this task multiple times.

Don't use the "reload" button during performance testing. Instead, launch the application from a bookmark. This simulates a user visiting the page from an external link or bookmark. In contrast, reloading the page forces the browser to send extra requests for cacheable resources which would not occur for a normal user.

With the correct performance testing approach in hand, you are ready to correctly assess the performance of Smart GWT. If you have followed Smart GWT best practices, you application will show a drastic reduction in dynamic requests due to features like:

- Adaptive Filtering and Sort: eliminates the most expensive category of search and sort operations by adaptively performing search and sort operations in-browser whenever possible.

  Adaptive Filter Example

  Adaptive Sort Example

- Central Write-Through Caching: smaller datasets can be centrally cached in-browser, even if they are modifiable

  DataSource.cacheAllData documentation

- Least Recently Used (LRU) Caching: automatic re-use of recently fetched results in picklists and other contexts.

Evaluating Interactive Performance

When evaluating interactive performance:

- Use GWT Compiled mode, *not* Hosted Mode

  Hosted mode can be slower than compiled mode by a difference of 10x or more.

- Disable Firebug or any similar third-party debugger or profiler

  These tools are great for debugging, but do degrade performance and can cause false memory leaks. End users won't have these tools enabled when they visit your application or site, so to assess real-world performance, turn these tools off.

- Close the Developer Console, revert log settings, and ensure Track RPCs is off

  Both refreshing the live Developer Console and storing large amounts of diagnostic output have a performance impact. To see the application as a normal end user, revert log settings to the default (only warnings are shown), disable "Track RPCs" in the RPC Tab, and close the Developer Console.

- Use normal browser cache settings

  Developers often set browsers to non-default cache settings, causing repeated requests that can degrade interactivity. End users won't have these special settings, so to assess real-world performance, revert to browser defaults.

For more performance testing tips and troubleshooting advice, see:

- The SmartGWT FAQ at:

  forums.smartclient.com

Evaluating Editions and Pricing

If you are a professionally employed developer, the cost of entry level commercial licenses is recouped if your team is able to leverage just one feature.

Consider, for example, the long term cost of recreating any single feature from the Pro product:

- time spent designing & developing your own version of the feature
- time spent testing & debugging your own version of the feature
- time spent addressing bugs in the feature after deployment
- time spent maintaining the code over time - supporting new browsers, or adding additional, related features that appear in the Pro product, that would have been effortless upgrades

If you work on a team, these costs may be multiplied many times as different developers repeatedly encounter situations where a feature from Pro would have saved effort.

Furthermore, looked at comprehensively, the cost of building and delivering an application includes time spent defining and designing the application, time spent developing, debugging and deploying the application, cost of the hardware the application runs on, licenses to other software, end user training, and many other costs.

The price of the most advanced Smart GWT technology is a tiny part of the overall cost of developing an application, and can deliver huge savings in all of these areas. For this reason, it makes sense to work with the most advanced Smart GWT technology available.

If you are a developer and you recognize that the features in Pro could save you time, you may find that an argument firmly based on cost savings and ROI (Return On Investment) will enable you to work with cutting edge technology and save you from wasting time "re-inventing the wheel."

A note on supporting Open Source

The free, open source (LGPL) version of Smart GWT exists because of the commercial version of the product. The free and commercial parts of the product are split in such a way that further development of the commercial version necessarily involves substantial upgrades to the open source version, and historically, new releases have contained as least as many new features in the free product as in the commercial version.

Further development of the commercial version also allows commercial features to migrate to the free, open source version over time.

As with any open source project, patches and contributions are always welcome. However, as a professionally employed developer, the **best** way to support the free product is to fuel further innovation by purchasing licenses, support, and other services.

Contacts

Isomorphic is deeply committed to the success of our customers. If you have any questions, comments, or requests, feel free to contact the Smart GWT product team:

Web	*www.smartclient.com*
General	*info@smartclient.com* *feedback@smartclient.com*
Evaluation Support	*forums.smartclient.com*
Licensing	*sales@smartclient.com*
Services	*services@smartclient.com*

We welcome your feedback, and thank you for choosing Smart GWT.

End of Guide