



SmartClient™ Quick Start Guide

SmartClient v10.0

SmartClient™ 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pursuant to a license agreement that contains use, duplication, disclosure and other restrictions; accordingly, it is “Unpublished-rights reserved under the copyright laws of the United States” for purposes of the FARs.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
Email: info@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, 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	iv
How to use this guide	vii
More than Just Widgets —A Complete Architecture	viii
Eliminates Cross-Browser Testing and Debugging.....	viii
Complete Solution	ix
Open, Flexible Architecture	ix
1. Overview.....	11
Architecture.....	11
Capabilities and Editions of SmartClient	12
Standard Capabilities	13
Optional Modules	14
SDK Components	15
2. Installation	16
Requirements	16
Steps	16
Server Configuration (optional)	19
3. Resources	20
Feature Explorer.....	20
Demo Application.....	21
Developer Console.....	22
Reference.....	27
Community Wiki	28
4. Coding	29
Languages.....	29
Headers	30
Components	31
Hello World	32
Deploying	33
5. Visual Components.....	34
Component Documentation & Examples.....	34
Identifying Components.....	35
Manual Layout.....	35
Drawing, Hiding, and Showing Components.....	37
Handling Events	37
6. Data Binding.....	39
Databound Components	39
Fields	40
Form Controls	42

DataSources.....	44
DataSource Operations	48
DataBound Component Operations	49
Data Binding Summary	50
7. Layout	52
Component Layout	52
Container Components	54
Form Layout	55
8. Data Integration	57
DataSource Requests.....	57
SmartClient Server Framework.....	58
DSRequests and DSResponses.....	59
Request and Response Transformation	59
Criteria, Paging, Sorting and Caching.....	61
Authentication and Authorization	62
Relogin.....	63
Binding to XML and JSON Services	64
WSDL Integration	66
9. SmartClient Server Framework.....	68
DataSource Generation	68
Server Request Flow.....	71
Direct Method Invocation	73
DMI Parameters	74
Adding DMI Business Logic	74
Returning Data	77
Queuing & Transactions	78
Queuing, RESTHandler, and SOAs	80
Operation Bindings	80
Declarative Security.....	82
Declarative Security Setup	84
Non-Crud Operations.....	85
Dynamic Expressions (Velocity)	87
Server Scripting	89
Including Values from Other DataSources	91
SQL Templating.....	93
SQL Templating — Adding Fields	96
Why focus on .ds.xml files?	98
Custom DataSources	99
Generic RPC operations (advanced)	101
10. Extending SmartClient	103
Client-side architecture	103
Customized Themes	104
Customized Components	106
New Components	107
New Form Controls	109
11. Tips.....	111
Beginner Tips	111
HTML and CSS Tips	111

Architecture Tips	112
12. Evaluating SmartClient.....	116
Which Edition to Evaluate	116
Evaluating Performance	117
Evaluating Interactive Performance	119
Evaluating Editions and Pricing.....	120
A note on supporting Open Source	121
Contacts	122

How to use this guide

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

- To have you working with SmartClient components and services in a matter of minutes.
- To provide a conceptual framework, with pointers to more detail, so you can explore SmartClient 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 SmartClient-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 SmartClient 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, markup, scripting), *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 SmartClient today.



If you can't wait to get started, you can skip directly to *Installation* (Chapter 2) to start a SmartClient development server and begin *Resources* (Chapter 3) and *Coding* (Chapter 4). But if you can spare a few minutes, we recommend reading the introductory chapters first, for the bigger picture of SmartClient goals and architecture.

Thank you for choosing SmartClient, and welcome.

Why SmartClient?

Smart Client 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 SmartClient platform that set it apart from other technologies with similar purpose.

More than Just Widgets —A Complete Architecture

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

The examples included with SmartClient 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.

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

Even if you adopt only part of the SmartClient 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 SmartClient platform has been designed with a clear understanding of the requirements you need to fulfill, **and**, the solutions built into SmartClient provide a “blueprint” for one way of meeting those requirements.

Eliminates Cross-Browser Testing and Debugging

SmartClient 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, SmartClient'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.

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

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

Complete Solution

SmartClient 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 SmartClient 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 SmartClient components and services into your HTML pages.

By giving you both options, SmartClient 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 SmartClient user interface components. SmartClient allows a smooth *evolution* of your existing web applications—you aren't forced to start over.

Next, SmartClient 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,

SmartClient provides flexible, generic interfaces to integrate with any data or service tier that you can access through Java code.

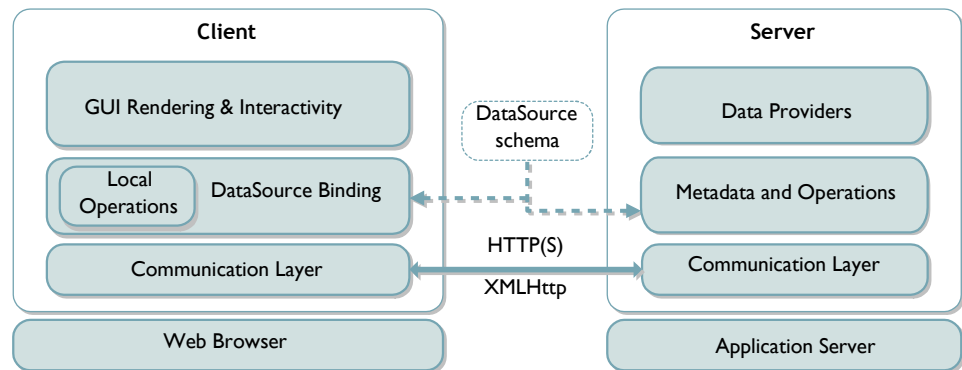
Finally, SmartClient is completely ***extensible***, all the way down to the web standards on which the system is constructed. If you can't do something "out of the box," you can build or buy components that seamlessly extend SmartClient in any manner you desire.

We welcome your comments and requests, however large or small, to feedback@smartclient.com.

1. Overview

Architecture

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



Within the web browser, SmartClient provides a deep stack of services and components for rich HTML5 / Ajax applications. For those using a Java-based server, SmartClient 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 SmartClient Ajax components, and server-side by SmartClient 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 SmartClient Java server components. However, if you do not use the SmartClient server components, all server-side functionality of DataSources must be implemented and maintained by your team.

Finally, note that SmartClient 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 SmartClient

SmartClient 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 SmartClient 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. Applications 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 12, [Evaluating SmartClient](#).

Standard Capabilities

The standard capabilities of the SmartClient web presentation layer include:

Area	Description
Foundation Services	SmartClient class system, data types, JavaScript extensions, and browser utilities.
Foundation Components	Building-block visual components, including <code>Canvas</code> , <code>Img</code> , <code>StretchImg</code> , and <code>StatefulCanvas</code> .
Event Handling	SmartClient event handling systems, including mouse, keyboard, focus, drag & drop, enable/disable, and selection capabilities.
Controls	Basic visual controls, including <code>Button</code> , <code>Toolbar</code> , <code>Menu</code> , and <code>Menubar</code> .
Forms	Form layout managers, value managers, and controls (including <code>TextItem</code> , <code>DateItem</code> , <code>CheckboxItem</code> , <code>SelectItem</code> , etc.).
Grids	<code>GridRenderer</code> , <code>ListGrid</code> and related subclasses, providing grid rendering, selection, sorting, editing, column handling, and cell events.
Trees	<code>Tree</code> data structures, and <code>TreeGrid</code> UI components, for managing hierarchical data.
Layout	Component layout managers and layout-managed containers, including <code>HLayout</code> , <code>VLayout</code> , <code>Window</code> , and <code>TabSet</code> .
Data Binding	Data model, cache management, and communication components including <code>DataSource</code> , <code>ResultSet</code> , and <code>RPCManager</code> .
Themes/Skins	Pervasive support and centralized control over theme/skin styles, images, and defaults, for personalization or branding.

Optional Modules

Isomorphic also develops and markets the following optional modules to extend the standard SmartClient system. For more information on these modules, see *SmartClient Reference* → *Optional Modules*.

Option	Description
SmartClient Server	Provides direct Java APIs for databinding and low level client-server communications, deep integration with popular Java technologies such as Spring, Hibernate, Struts, and others. Extensive server-side validators that match the client-side versions and work automatically. For more information, please see the “SmartClient Server Feature Summary” in the Concepts section of the SmartClient Reference.
Analytics	Multi-dimensional data binding and interactive <i>CubeGrid</i> components (cross-tabs, dynamic data loading, drag-and-drop pivoting).
Real-Time Messaging	Real-time, server push messaging over HTTP, with Java Message Server (JMS) backed publish and subscribe services.
Network Performance	File packaging, caching, and compression services for optimal performance of distributed applications.

SDK Components

The SmartClient Software Developer Kit (SDK) includes extensive documentation and examples to accelerate you along the learning curve. These resources are linked from the SDK Explorer, and are available in the `docs/` and `examples/` directories of your SDK distribution.

The SmartClient SDK also provides the following supplementary, develop-time components for rapid development:

Development Component	Component Description
Developer Console	Provides client-side application debugging, inspection, and profiling.
Admin Console	Provides a browser-based UI for server configuration and datasource management. Note: Requires SmartClient Server.
Embedded server (Tomcat)	Enables a lightweight, stand-alone development environment.
Embedded database (HSQLDB)	Provides a basic persistence layer for rapid development. Note: Requires SmartClient Server.
Object-relational connector (JDBC/ODBC)	Enables rapid development of your presentation layer against a relational database, prior to (or in parallel with) development of your server-side business logic bindings. Note: Requires SmartClient Server.

The SmartClient SDK provides direct database support for lightweight application development purposes only. Production SmartClient applications are typically bound to application-specific data objects (EJBs, POJOs), web services, email and IM servers, and structured data feeds.



Chapter 8 ([Data Integration](#)) outlines the integration layers and interfaces for your production data and services.

2. Installation

Requirements

To get started quickly, we will use the embedded application server (Apache Tomcat 5.0) and database engine (HSQL DB 1.7) that are included in the SmartClient SDK distribution.

Your only system requirements in this case are:

- **Java SDK (JDK)** v1.4+ (you can download JDK 1.5/5.0 from <http://java.sun.com/j2se/1.5.0/download.jsp>)
- a **web browser** to view SmartClient examples and applications (see `docs/readme.html` in the SDK for a complete list of supported browsers and versions)
- a **text editor** to create and edit SmartClient code and examples

If you wish to install SmartClient in a different application server and/or run the SDK examples against a different database, please see `docs/installation.html` in the SDK. For purposes of this Quick Start, we strongly recommend using the embedded server and database. You can always redeploy and configure your SmartClient SDK in another application server later.

Steps

To install and start your SmartClient development environment:

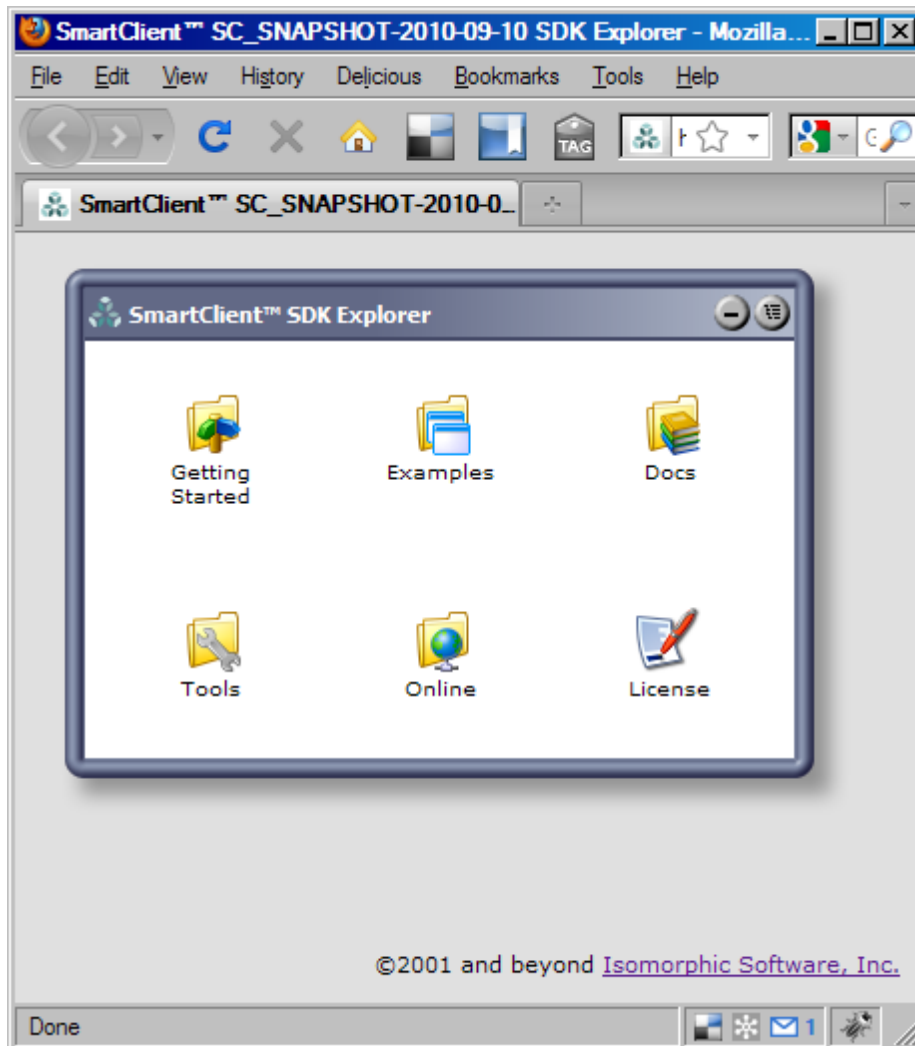
1. Download and install JDK 1.4+ if necessary (Mac OS X users note: a JDK is pre-installed on your system)
2. Start the embedded server by running
`start_embedded_server.bat` (Windows), `.command` (Mac OS X),
or `.sh` (*nix)

3. Open the `open_ISC_SDK_from_server` shortcut (Windows/MacOS) or open a web browser and browse to `http://localhost:8080/index.html` (all systems)

Depending on your system configuration, you may need to perform one or more additional steps:

- *If you already have a JDK or JRE installed on your system,* you may need to set a `JAVA_HOME` environment variable pointing to the home directory of JDK 1.4+, so the server will use the correct version of Java.
- *If port 8080 is already in use on your system,* you may specify a different port for the embedded server by appending
`--port newPortNum` (e.g. `--port 8081`) to the `start_embedded_server.bat`, `.command`, or `.sh` command. If you do change the default port, you must browse directly to `http://localhost:newPortNum/index.html`. to open the SDK Explorer
- *If your web browser is configured to use a proxy server,* you may need to bypass that proxy for local addresses. In Internet Explorer, go to Tools → Internet Options... → Connections → LAN Settings..., and check “Bypass proxy server for local addresses”. In Firefox, go to Tools → Options... → General → Connection Settings... and enter “localhost” in the “No Proxy for” field.

When you have successfully started the server and opened `http://localhost:8080/index.html` in your web browser, you should see the SmartClient SDK Explorer:



Instructions for adding SmartClient to any existing Java web project in any IDE are in the *SmartClient Reference - Concepts > Deploying SmartClient*.

Server Configuration (optional)

You do not need to perform any server configuration for this Quick Start. However, for your information:

- The *SmartClient Admin Console* (linked from the SDK Explorer) provides a graphical interface to configure direct database connections, create database tables from DataSource descriptors, and import test data. Note: Requires SmartClient Server.
- Other server settings are exposed for direct configuration in:

```
WEB-INF/classes/server.properties  
WEB-INF/web.xml
```

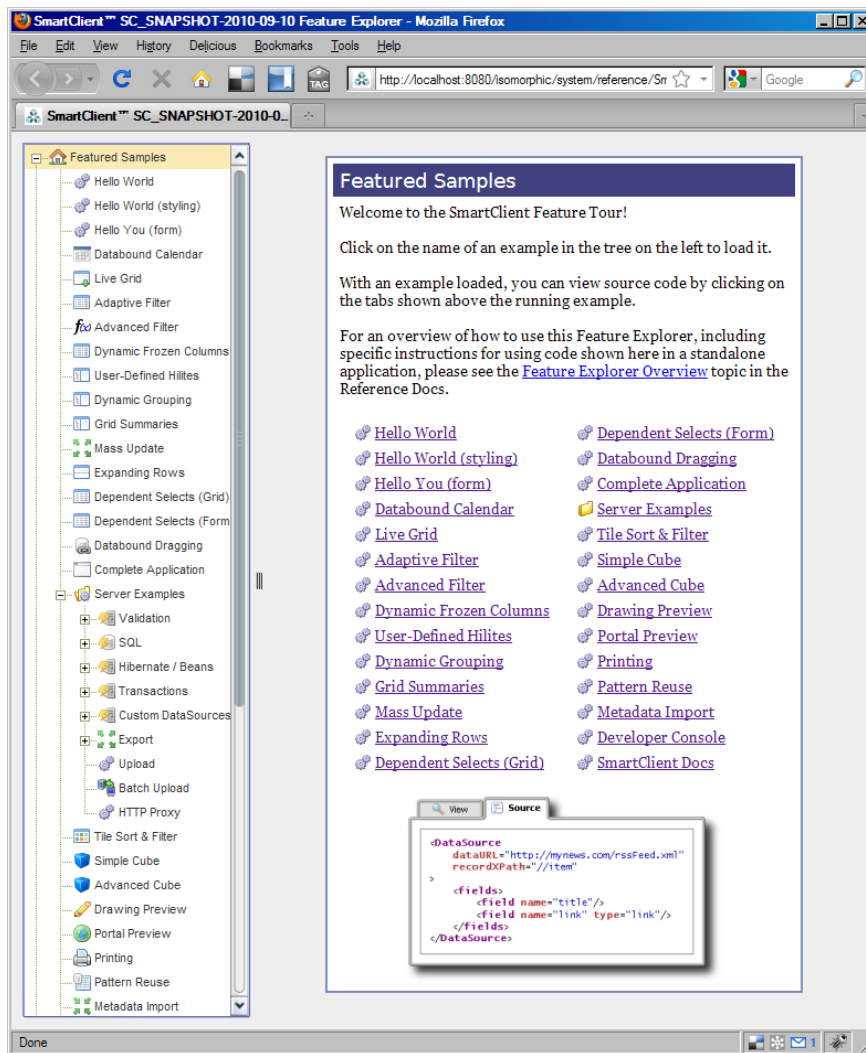


If you have any problems installing or starting SmartClient, try the SmartClient Developer Forums at forums.smartclient.com.

3. Resources

Feature Explorer

From the SmartClient SDK Explorer, pick **Getting Started** then **Feature Explorer**. When the Feature Explorer has loaded, you should see the following screen:



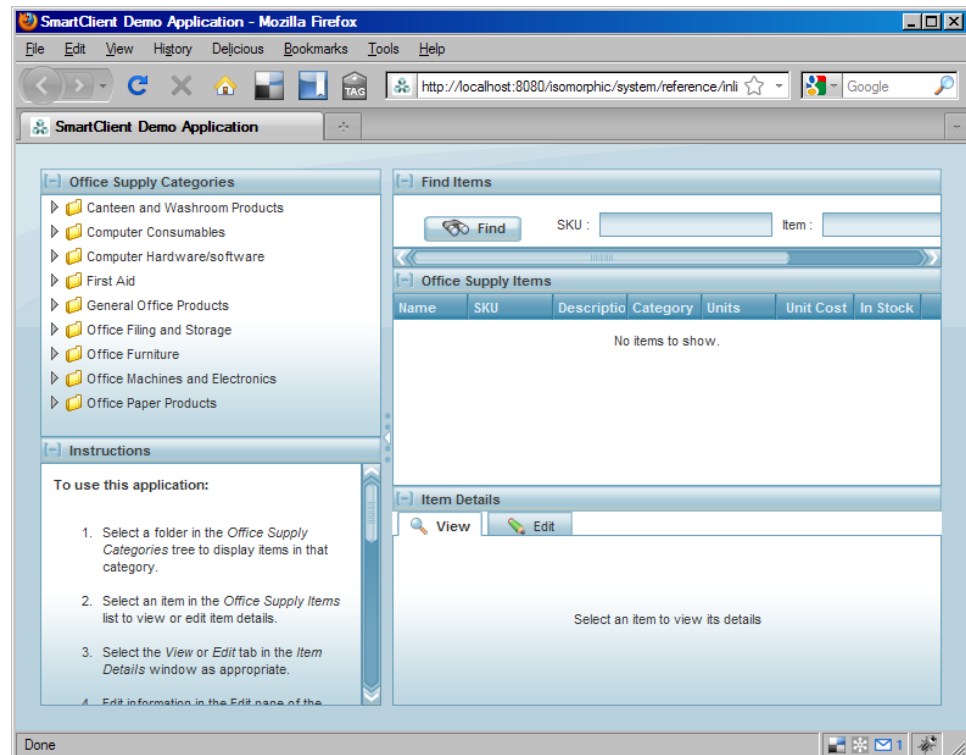
The Feature Explorer is your best starting point for exploring SmartClient capabilities and code.

The code for the examples in the Feature Explorer can be edited within the Feature Explorer itself, however, changes will be lost on exit. To create a permanent, standalone version of an example found in the Feature Explorer, copy the source code into one of the templates in the `templates/` directory (discussed in more detail in *Headers* of the next chapter, *Coding*).

All of the resources used by the Feature Explorer are also available in the `isomorphic/system/reference/` directory. In particular, `exampleTree.xml` establishes the tree of examples on the left hand side of the Feature Explorer interface, and contains paths to example files in the `inlineExamples/` subdirectory. Note that some `DataSources` shared by multiple examples are in the central `shared/ds` and `examples/shared/ds` directories.

Demo Application

From the SmartClient SDK Explorer, pick **Getting Started** then **Demo App**. The first launch of this application will take several seconds, as the application server parses and compiles the required files. When the application has loaded, you should see the following screen:



This example application demonstrates a broad range of SmartClient user interface, data binding, and layout features.

To experience this application as an end user, follow the steps in the **Instructions** window at the bottom left of the application window.

The SmartClient SDK provides two versions of the code for this application, one in JavaScript and one in XML, to demonstrate alternate coding formats.



SmartClient JS and XML coding formats are discussed in detail in Chapter 4 ([Coding](#))

To explore the application code for this application, click on the **XML** or **JS** links underneath the Demo App icon in the SDK Explorer. You can also view and edit the source code for this application directly from the `isomorphic/system/reference/inlineExamples/demoApp/` directory in the SDK. After you make changes to the code, simply reload the page in your web browser to see the results.

Each `.jsp` file in the `demoApp/` directory contains all component definitions and client-side logic for the application. The only other source files for this application are `demoApp_helpText.js` and `demoApp_skinOverrides.js` in the same directory, and the two datasource descriptors in:

```
examples/shared/ds/supplyItem.ds.xml
examples/shared/ds/supplyCategory.ds.xml
```

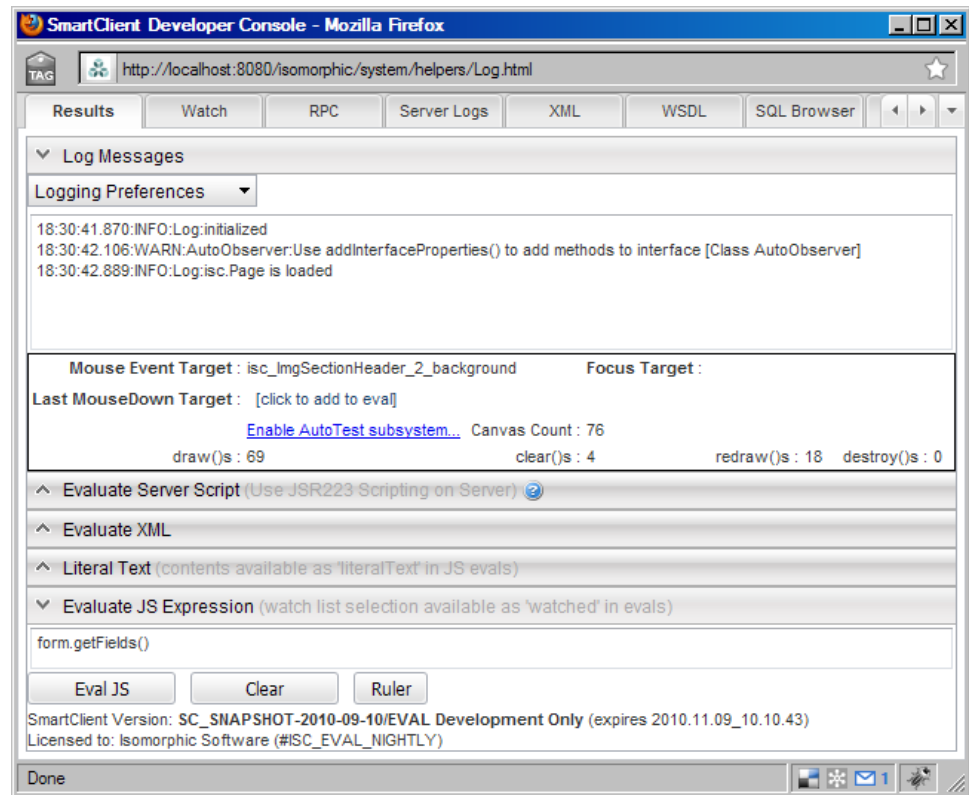
The key concepts underlying this application—SmartClient JS and XML Coding, Visual Components, DataSources, and Layouts—are covered in chapters 4 through 8 of this guide. You may want to briefly familiarize yourself with the code of this example now, so you can refer back to the code to ground each concept as it is introduced.

Developer Console

The SmartClient Developer Console is a suite of development tools implemented in SmartClient 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 systems
- runtime code inspection and evaluation
- runtime component inspection
- tracing and profiling
- integrated reference docs

You can open the Developer Console from any SmartClient-enabled page by typing `javascript:isc.showConsole()` in the address bar of your web browser. Try it now, while the demo application is open in your browser. The following window will appear:



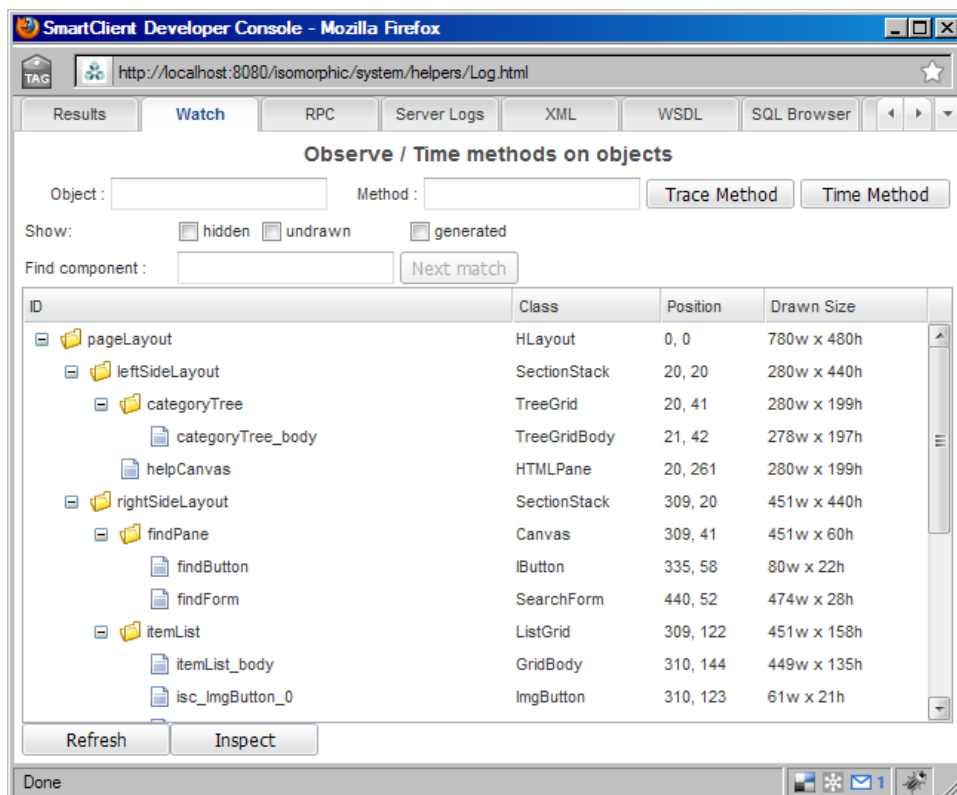
Popup blocker utilities may prevent the Developer Console from appearing, or browser security settings may disallow JavaScript URLs. Holding the **Ctrl** key while opening the console will bypass most popup blockers. Creating a bookmark for a JavaScript URL will allow you to execute it by clicking on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

The **Results** pane of the Developer Console displays:

- Messages logged by SmartClient or your application code through the SmartClient 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.
- SmartClient 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. For example, try mousing over the instructions area for the demo application; you should see “helpCanvas” as the Current Event Target.
- A runtime code evaluation area. You may evaluate expressions and execute actions from this area. For example, with the demo application running, try evaluating each of these expressions:

```
categoryTree.getSelectedRecord()
helpCanvas.hide()
helpCanvas.show()
```

The **Watch** pane of the Developer Console displays a tree of SmartClient user interface components in the current application. With the demo application running, this pane appears as follows:

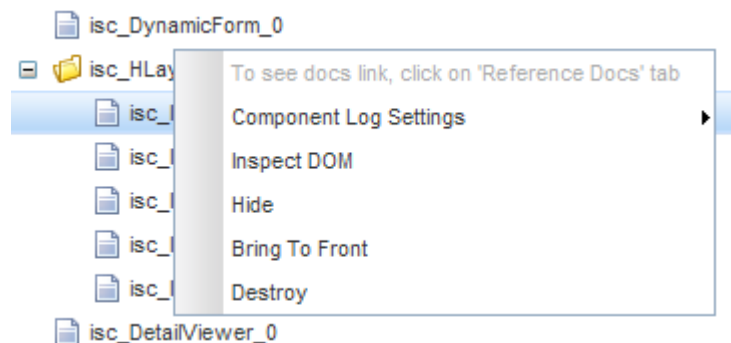


In the **Watch** pane, you may:

- 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, including a direct link to the API reference for that component's class.



- Right-click on the column headers of the tree to show or hide columns.

The Developer Console is an essential tool for all SmartClient application developers and should be open whenever you are working with SmartClient. 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 window at any time.

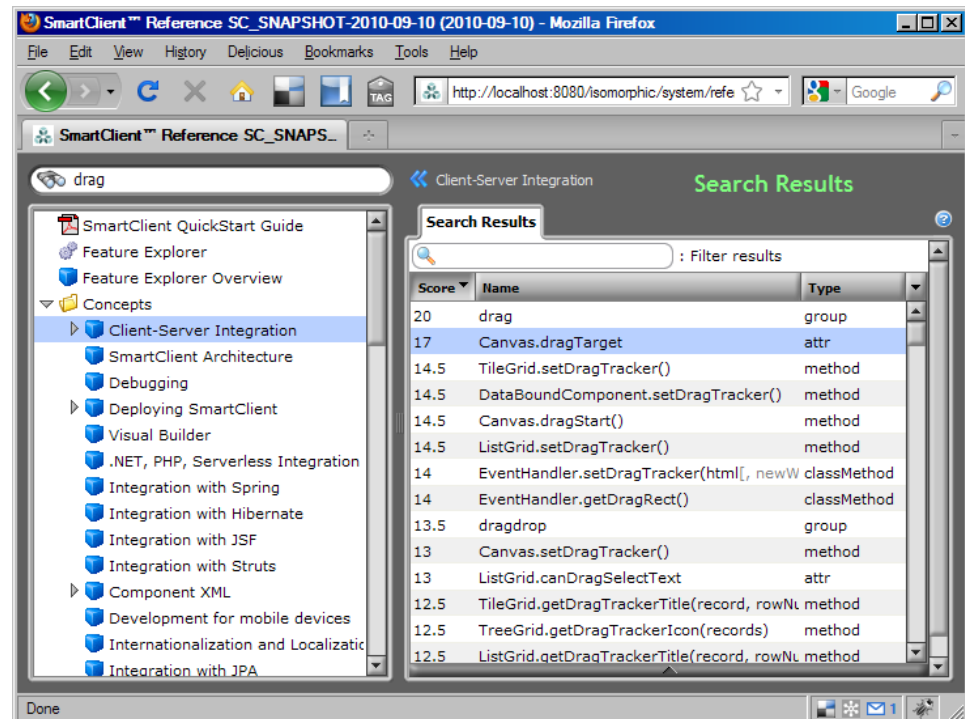
If you have shown the console for a SmartClient application in one browser window, and then open an application in another browser window, you must close the console before you can show it from the new window.

Reference

The core documentation for SmartClient is the *SmartClient Reference*, an interactive reference viewer implemented in SmartClient. You may access the *SmartClient Reference* in any of the following ways:

- from the **Reference Docs** tab of the Developer Console
- by right-clicking on a component in the **Watch** tab of the Developer Console, and selecting “Show doc for...”
- from the **SmartClient Reference** icon in SDK Explorer → Docs → SmartClient Reference
- from the `docs/SmartClient_Reference.html` launcher in the SDK

The *SmartClient Reference* provides integrated searching capabilities. Enter your search term in the field at top-left, then press Enter. The viewer will display a list of relevance-ranked links. For example, searching on “drag” generates the following results:



If you are new to SmartClient, you may want to read the conceptual topics in the *SmartClient Reference* for more detail after completing this Quick Start guide. These topics are indicated by the blue cube icon (📦) in the reference tree.

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/>

4. Coding

Languages

SmartClient applications may be coded in:

- XML—for declarative user interface and/or datasource definitions – development in this format requires the SmartClient Server.
- JavaScript (JS) —for client-side user interface logic, custom components, and procedural user interface definitions
- Java—for data integration when using the SmartClient Java Server

SmartClient provides multiple layers of structure and services on top of the JavaScript language, including a real class system, advanced data types, object utilities, and other language extensions. The structure of SmartClient JS code is therefore more similar to Java than it is to the free-form JavaScript typically found in web pages.

To define user interface components, you may use either SmartClient XML or SmartClient JS. Both formats have their merits:

SmartClient XML

- more tools available for code validation
- more familiar to HTML programmers
- forces better separation of declarative UI configuration, and procedural UI logic

SmartClient JS

- more efficient
- easier to read when declarative and procedural code must be combined
- works in stand-alone examples (no server)
- allows programmatic (runtime) component instantiation

Each format also has its quirks: In JS, missing or dangling commas are a common cause of parsing errors. In XML, quoting and escaping rules can make code difficult to read and write.



Isomorphic currently recommends using JavaScript (JS) to define your SmartClient user interface components, for maximum flexibility as your applications evolve. However, the SmartClient SDK provides examples in both JS and XML. You can decide which is appropriate for your style and your specific needs.

If you are new to JavaScript, you will need to be aware that:

- JavaScript identifiers are case-sensitive. For example, `Button` and `button` refer to different entities. SmartClient component class names (like `Button`) are capitalized by convention.
- JavaScript values are not strongly typed, but they *are* typed. For example, `myVar=200` sets `myVar` to the number 200, while `myVar="200"` sets `myVar` to a string.

Headers

Every SmartClient application is launched from a web page, which is usually called the *bootstrap* page. In the header of this page, you must load the SmartClient client-side engine, specify a user interface “skin,” and configure the paths to various SmartClient resources.

The exact format of this header depends on the technology you use to serve your bootstrap page. The minimal headers for loading a SmartClient-enabled `.jsp` or `.html` page are as follows.

Java server (.jsp)

```
<%@ taglib uri="isomorphic" prefix="isomorphic" %>
<HTML><HEAD>
  <isomorphic:loadISC skin="SmartClient"/>
</HEAD><BODY>
```

Generic web server (.html)

```
<HTML><HEAD>
<SCRIPT>var isomorphicDir="../isomorphic/";</SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_Core.js></SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_Foundation.js></SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_Containers.js></SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_Grids.js></SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_Forms.js></SCRIPT>
<SCRIPT SRC=../isomorphic/system/modules/ISC_DataBinding.js></SCRIPT>
<SCRIPT SRC=../isomorphic/skins/SmartClient/load_skin.js></SCRIPT>
</HEAD><BODY>
```

If you use the `isomorphic:loadISC` tag (available in `.jsp` pages only), SmartClient will automatically detect and set the appropriate file paths. If you use the generic header (which will work in any web page), you may need to change the file paths to locate the `isomorphic/` directory. This example assumes that the bootstrap page is located in a directory that is adjacent to the `isomorphic/` directory.

Note that both examples above load all standard modules. Your application may need only some modules, or may also load the optional modules discussed in Chapter 1.



The SmartClient SDK provides complete `.jsp` and `.html` template pages in the top-level `templates/` directory, for easy integration with your development environment.



For information about switching to a different skin or using a custom skin, see the *Customized Themes* section in Chapter 9, *Extending SmartClient*.

Components

SmartClient is an object-oriented system. You assemble your web application GUIs from SmartClient *components*. These components are defined as reusable *classes*, from which you create specific *instances*. Component classes and instances provide *properties* (aka *attributes*) that you can set at initialization, and *methods* (aka *functions*) that you can call at any time in your client-side logic.

You use the `create()` method to instantiate SmartClient components in JS code. This method takes as its argument a JavaScript *object literal*—a collection of comma-delimited `property:value` pairs, surrounded by curly braces. For example:

```
isc.Button.create({title:"Click me", width:200})
```

For better readability, you can format your component constructors with one property per line, as follows:

```
isc.Button.create({
  title: "Click me",
  width: 200
})
```



The most common syntax errors in JS code are missing or dangling commas in object literals. If you omit the comma after the `title` value in the example above, the code will not parse in any web browser. If you include a comma following the `width` value, the code will not parse in Internet Explorer. SmartClient scans for dangling commas and will log this common error to your server output (visible in the terminal window where you started the server), for easier debugging.

To create a SmartClient component in XML code, you create a tag with the component's class name. You can set that component's properties either as tag attributes:

```
<Button
  title="Click me"
  width="200"
/>
```

or in nested tags:

```
<Button>
  <title>Click me</title>
  <width>200</width>
</Button>
```

The latter format allows you to embed JS inside your XML code, e.g., for dynamic property values, by wrapping it in `<JS>` tags:

```
<Button>
  <title>
    <JS>myApp.i18n.clickMe</JS>
  </title>
  <width>200</width>
</Button>
```

At the page level, SmartClient XML code must be wrapped in `<isomorphic:XML>` tags—see below for an example.

Hello World

The following examples provide the complete code for a SmartClient “Hello World” page, in three different but functionally identical formats.

Try recreating these examples in your editor. You can save them in the `examples/` directory of the SmartClient SDK, with the appropriate file extensions (`.html` or `.jsp`).

helloworld.jsp (SmartClient JS)

```
<%@ taglib uri="isomorphic" prefix="isomorphic" %>
<HTML>
  <HEAD>
    <isomorphic:loadISC skin="standard"/>
  </HEAD>
  <BODY>
    <SCRIPT>
      isc.Button.create({ title:"Hello",
        click:"isc.say('Hello World')",
      })
    </SCRIPT>
  </BODY>
</HTML>
```


helloworldXML.jsp (SmartClient XML)

```
<%@ taglib uri = "isomorphic" prefix = "isomorphic" %>
<HTML>
  <HEAD>
    <isomorphic:loadISC skin = "standard"/>
  </HEAD>
  <BODY>
    <SCRIPT>
      <isomorphic:XML>
        <Button title = "Hello" click = "isc.say('Hello World')" />
      </isomorphic:XML>
    </SCRIPT>
  </BODY>
</HTML>
```

helloworld.html (SmartClient JS)

```
<HTML>
  <HEAD>
    <SCRIPT>var isomorphicDir = "../isomorphic/";</SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_Core.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_Foundation.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_Containers.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_Grids.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_Forms.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/system/modules/ISC_DataBinding.js></SCRIPT>
    <SCRIPT SRC = ../isomorphic/skins/SmartClient/load_skin.js></SCRIPT>
  </HEAD>
  <BODY>
    <SCRIPT>
      isc.Button.create({
        title: "Hello",
        click: "isc.say('Hello World')"
      })
    </SCRIPT>
  </BODY>
</HTML>
```

You can open the .html version directly from your file system (by double-clicking the file's icon), provided your browser allows interactive web pages to run from your file system.

You must open the .jsp versions through your server, as follows:

```
http://localhost:8080/examples/helloworld.jsp
http://localhost:8080/examples/helloworldXML.jsp
```



These examples are also provided in the top-level `templates/` directory—but we highly recommend creating them yourself for the learning experience.

Deploying

For instructions on deploying a SmartClient application, see:

- [SmartClient Reference – Concepts > Deploying SmartClient](#)

The next chapter explains how to configure and manipulate SmartClient visual components in more detail.

5. Visual Components

SmartClient provides two families of visual components for rich web applications:

- **Independent visual components**, which you will 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 6, [Data Binding](#), and especially Chapter 7, [Layout](#).

Component Documentation & Examples

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



SmartClient Reference – For component interfaces (APIs), see *Client Reference*. Form controls are sub-listed under *Client Reference > Forms > Form Items*.



Component Code Examples – For live examples of component usage, see the SmartClient Feature Explorer (*Examples → Feature Explorer* in the SDK Explorer, or http://localhost:8080/isomorphic/system/reference/SmartClient_Explorer.html from a running SmartClient server).



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 Chapters 6 ([Data Binding](#)) and 7 ([Layout](#)), respectively.

Identifying Components

You can identify SmartClient components by setting their `ID` property:

```
isc.Label.create({
  ID: "helloWorldLabel",
  contents: "Hello World"
})
```

By default, component IDs are created in the global namespace, so your client-side code may reference `helloWorldLabel` to manipulate the `Label` instance created above. You should assign unique IDs that are as descriptive as possible of the component's type or purpose. Some common naming conventions are:

- include the component's type (such as `button` or `btn`)
- include the component's action (such as `update`)
- include the datasource the component affects (such as `salesOrder`). For example, `salesOrderUpdateBtn`

You can alternatively manage your components by saving the internal reference that is returned from the `create()` call. For example,

```
var helloWorldLabel = isc.Label.create({
  contents: "Hello World"
});
```

In this case, a unique ID will be assigned to the component. The current internal format for auto-assigned IDs is `isc_ClassName_ID_#`.

Manual Layout

You can configure and manipulate SmartClient components by setting component properties and calling component methods. The most basic properties for a visual component involve its position, size, and overflow:

- `left`
- `top`
- `width`
- `height`
- `overflow`
- `position`

`left` and `top` take integer values, representing a number of pixels from the top-left of the component's container (typically a web page, `Layout`, `Window`, or `TabSet`). `width` and `height` take integer pixel values (default 100 for most classes), and can also take string percentage values (e.g. "50%"). For example:

```
isc.Label.create({
  left: 200, top: 200,
  width: 10,
  contents: "Hello World"
})
```

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 need 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), `"scroll"` (always show scrollbars), or `"auto"` (show scrollbars only when needed). For example:

```
isc.Label.create({
  left: 200, top: 200,
  width: 20,
  contents: "Hello World",
  overflow: "hidden"
})
```

By default, SmartClient visual components are positioned at absolute pixel coordinates in their containers. If you need to embed a component in the flow of existing HTML, you may set its `position` property to `"relative"`. For example:

```
<LI>first item</LI>
<LI>
  <SCRIPT>
    isc.Button.create({
      title: "middle item",
      position: "relative"
    })
  </SCRIPT>
</LI>
<LI>last item</LI>
```



If you work directly with HTML or CSS code, you must test your code on all supported browsers for inconsistencies.

In particular, the same HTML and CSS layout code can produce many different results in different browsers, browser versions, and DOCTYPE modes. Whenever possible, you should consider using SmartClient components and layouts to insulate you from browser-specific interpretations of HTML and CSS.

In most applications, you will want more flexible, dynamic layout of your visual components. Chapter 7 (*Layout*) introduces the SmartClient Layout managers, which you can use to automatically size, position, and reflow your components at runtime.

Drawing, Hiding, and Showing Components

In a SmartClient-enabled application, you may load hundreds of user interface components in the bootstrap page, and then navigate between views on the client by hiding and showing these components. The basic APIs for hiding and showing components are:

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

The `autoDraw` property defaults to `true`, so a component is usually shown as soon as you `create()` it. Set `autoDraw` to `false` to defer showing the component. For example:

```
isc.Button.create({
  ID: "hiddenBtn",
  title: "Hidden",
  autoDraw: false
})
```

To show this button:

1. Open the SmartClient Developer Console from the page that has created the button.
2. Type `hiddenBtn.show()` in the JS evaluation area.
3. Click the “Eval” button to execute that code.



For more information on architecting your applications for high-performance, client-side view navigation, see *SmartClient Reference* → *Concepts* → *SmartClient Architecture*.

Handling Events

SmartClient 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 common SmartClient component event handlers include:

- `click` (for buttons and menu items)
- `recordClick` (for listgrids and treegrids)
- `change` (for form controls)
- `tabSelected` (for tabsets)

Component event handlers are set using a special type of property called a *string method*. These properties may be specified either as:

- a *string* of JavaScript to evaluate when the event occurs; or
- a JavaScript *function* to call when the event occurs

For example:

```
isc.Button.create({
  ID: "clickBtn",
  title: "click me",
  click: "isc.warn('button was clicked')"
})
```

Is functionally identical to:

```
isc.Button.create({
  ID: "clickBtn",
  title: "click me",
  click: function () {
    isc.warn('button was clicked');
  }
})
```

For event handling in applications, you can set your event handlers to strings that execute external functions. This approach enables better separation of user interface structure and logic:

```
isc.Button.create({
  ID: "clickBtn",
  title: "click me",
  click: "clickBtnClicked()"
})

function clickBtnClicked() {
  isc.warn('button was clicked');
}
```



For more information on available SmartClient events, see:

- *SmartClient Reference* – Component-specific APIs under *Client Reference*
- *SmartClient Reference* – EventHandler APIs under *Client Reference* → *System* → *EventHandler*

6. Data Binding

Databound Components

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

Visual Component	Display Data	Query Data	Edit Data
DynamicForm	✓		✓
ListGrid	✓	✓	✓
TreeGrid	✓	✓	✓
CubeGrid (Analytics option)	✓	✓	
DetailView	✓		
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 8 (*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 title, width, alignment). Component fields are discussed immediately below.
- **DataSource** fields provide **metadata** describing the objects in a particular datasource (such as data type, length, required). DataSource fields are discussed in [DataSources](#).

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
CubeGrid (Analytics option)	facets (row & column headers)
DetailView	rows
TileGrid	rows within tiles
Calendar	event duration and description

You can specify the displayed fields of a visual component via the `fields` property, which takes an array of field definition objects. For example:

```
isc.ListGrid.create({
  ID: "contactsList",
  left: 50, top: 50,
  width: 300,
  fields: [
    {name: "salutation", title: "Title"},
    {name: "firstname", title: "First Name"},
    {name: "lastname", title: "Last Name"}
  ]
})
```


Try reproducing this example. When you load it in your web browser, you should see a ListGrid that looks like this:

Title	First Name	Last Name	
No items to show.			

The `name` property of a field is the special key that connects that field to actual data values. For a simple `ListGrid` or `DetailView`, you can specify data values directly via the `data` property, which takes an array of record objects. Add this code to the `ListGrid` definition above (remembering to add a comma between the `fields` and `data` properties):

```
data: [
  {salutation:"Ms",  firstname:"Kathy",  lastname:"Whitting"},
  {salutation:"Mr",  firstname:"Chris",  lastname:"Glover"},
  {salutation:"Mrs", firstname:"Gwen",   lastname:"Glover"}
]
```

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 (directly setting `data`) is appropriate mainly for lightweight, read-only uses (i.e., 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 set the `dataSource` property instead to bind them to `DataSource` objects. See *DataSources* for details.

The basic field definitions in the `ListGrid` above are reusable across components. For example, you could copy these field definitions to create a `DynamicForm`:

```
isc.DynamicForm.create({
  ID: "contactsForm",
  left: 50, top: 250,
  width: 300,
  fields: [
    {name:"salutation", title:"Title"},
    {name:"firstname", title:"First Name"},
    {name:"lastname", title:"Last Name"}
  ]
})
```

which will display as:

Title :

First Name :

Last Name :



For complete documentation of component field properties (presentation attributes), see:

- *SmartClient Reference – Client Reference → Forms → Form Items* (all entries)
- *SmartClient Reference – Client Reference → Grids → ListGrid → ListGridField*

`DataSource` field properties (data attributes) are discussed in *DataSources*.

Form Controls

Field definitions also determine which *form controls* are presented to users, for editable data values in forms and grids. You can specify the form control to use for a field by setting its `editorType` property.

The default `editorType` is "text", which displays a simple text box editor. This control is an instance of the `TextItem` class.

If a component is bound to a `DataSource`, it will automatically display appropriate form controls based on attributes of its `DataSource` fields (e.g. checkbox for boolean values, date picker for date values, etc). However, there may be more than one way to present the same value. For example, a dropdown control (`selectItem`) and a set of radio buttons (`radioGroupItem`) are both appropriate for presenting a relatively small set of values in a form.

To override the default form control for a field, set `editorType` to the class name for that control, in lower case, minus the "Item". For example, for a `CheckboxItem`, you can set `editorType:"checkbox"`.

The following code extends the previous `DynamicForm` example to use an assortment of common form controls, specified by `editorType`:

```
isc.DynamicForm.create({
  ID: "contactsForm", left: 50, top: 250, width: 300,
  fields: [
    {name:"salutation", title:"Title", editorType: "select",
      valueMap:["Ms", "Mr", "Mrs"]
    },
    {name:"firstname", title:"First Name"},
    {name:"lastname", title:"Last Name"},
    {name:"birthday", title:"Birthday", editorType:"date"},
    {name:"employment", title:"Status", editorType:"radioGroup",
      valueMap:["Employed", "Unemployed"]
    },
    {name:"bio", title:"Biography", editorType:"textArea"},
    {name:"followup", title:"Follow up", editorType:"checkbox"}
  ]
})
```

This form will appear as follows:

Title :

First Name :

Last Name :

Birthday :

Status : ☐ Employed ☐ Unemployed

Biography :

☐ Follow up



For more information on the layout of managed form controls, see “Form Layout” in Chapter 7 ([Layout](#)).

DataSourcees

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

- Separate your data model attributes from your presentation attributes.
- Share your data models across multiple applications and components, and across both client and server.
- Display and manipulate persistent data and data-model relationships (e.g. 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 directly in XML or JS format, or can be created dynamically from existing metadata (for more information, see **SmartClient Reference** → *Client Reference* → *Data Binding* → *DataSource* → *Creating DataSourcees*). The XML format is interpreted and shared by both client and server, while the JS format is used by the client only. Note that use of the XML format requires the optional SmartClient Server.

There are four basic rules to creating 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. Appending “DS” to the ID is a good convention to easily identify DataSource references in your code.
2. Specify a field element with a unique `name` (in this DataSource) for each field that will be exposed to the presentation layer.
3. Specify a `type` attribute on each field element (see below 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 DataSourcees, 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 11, [Tips](#).

Following these rules, a DataSource descriptor for the “contacts” example earlier in this chapter looks like:

```
<DataSource ID="contactsDS">
  <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>
```

For your convenience, this descriptor is already saved in `shared/ds/contactsDS.ds.xml`. Note that this code is the entire content of the file—there are no headers, `<HTML>` tags, or other wrappers around the DataSource descriptor.



Every DataSource field must specify a `type`, and editable DataSources (i.e., supporting Add, Update, or Remove operations) must specify exactly one field with `primaryKey="true"`.



For more information on defining, creating, and locating DataSources, see *SmartClient Reference* → *Client Reference* → *Data Binding* → *DataSource*. The *Creating DataSources* and *Client Only DataSources* subtopics provide additional detail.

To load this DataSource in previous “contacts” example, add the following tag inside the `<SCRIPT>` tags, before the ListGrid and DynamicForm components are created:

```
<isomorphic:loadDS ID="contactsDS" />
```

Now the components can reference this shared DataSource via their `dataSource` properties, instead of specifying `fields`. The complete code for a page that binds a grid and form to this DataSource is:

```
<%@ taglib uri="isomorphic" prefix="isomorphic" %>
<HTML><HEAD>
  <isomorphic:loadISC />
</HEAD><BODY>
  <SCRIPT>
    <isomorphic:loadDS ID="contactsDS" />
    isc.ListGrid.create({
      ID: "contactsList",
      left: 50, top: 50,
      width: 500,
      dataSource: contactsDS
    });
    isc.DynamicForm.create({
      ID: "contactsForm",
      left: 50, top: 200,
      width: 300,
      dataSource: contactsDS
    });
  </SCRIPT>
</BODY></HTML>
```

This example entirely replaces `fields` with a `dataSource` for simplicity. However, these two properties will usually co-exist on your databound components. The component field definitions in `fields` specify presentation attributes, while the DataSource field definitions specify data attributes (see table below).

SmartClient merges your component field definitions and DataSource field definitions based on the `name` property of the fields. By default, the order and visibility of fields in a component are determined by the `fields` array. To change this behavior, see `useAllDataSourceFields` in the *SmartClient Reference*.

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.
length	maximum length of text value in characters
hidden	true; whether this field should be entirely hidden from the end user. It will not appear in the default presentation, and it will not appear in any field selectors (e.g. the column picker menu in a ListGrid) available to the end user.
required	true false
valueMap	an array of values, or an object containing storedValue:displayValue pairs
primaryKey	true; 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 11, Tips .
foreignKey	a reference to a field in another DataSource (for example, dsName.fieldName)
rootValue	for fields that establish a tree relationship (by foreignKey), this value indicates the root node of the tree



For complete documentation of the metadata properties supported by SmartClient DataSources and components, see *SmartClient Reference* → *Client Reference* → *Data Binding* → *DataSource* → *DataSourceField*.



For DataSource usage examples, see the descriptors in `examples/shared/ds/`. These DataSources are used in various SmartClient SDK examples, including the SmartClient Feature Explorer



For an example of a DataSource relationship using `foreignKey`, see `examples/databinding/tree_databinding.jsp` (TreeGrid UI) and `shared/ds/employees.ds.xml` (associated DataSource).

As mentioned under “Form Controls” above, databound components will automatically display appropriate form controls based on attributes of their DataSource fields. The rules for this automatic selection of form controls are:

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 this automatic behavior by explicitly setting `editorType` on any component field.

DataSource Operations

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

Operation	Methods	Description
Fetch	<code>fetchData (...)</code>	retrieves records from the datasource that exactly match the provided criteria
	<code>filterData (...)</code>	retrieves records from the datasource that contain (substring match) the provided criteria
Add	<code>addData (...)</code>	creates a new record in the datasource with the provided values
Update	<code>updateData (...)</code>	updates a record in the datasource with the provided values
Remove	<code>removeData (...)</code>	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, etc. (see `DSRequest` in the *SmartClient Reference* for details)

You may call any of these five methods directly on a `DataSource` object, or on a databound `ListGrid` or `TreeGrid`. For example:

```
contactsDS.addData(
  {salutation:"Mr", firstname:"Steven", lastname:"Hudson"},
  "say(data[0].firstname + 'added to contact list')",
  {prompt:"Adding new contact..."}
);
```

or

```
contactsList.fetchData(
  {lastname:"Glover"}
);
```



DataSource operations will only execute if the DataSource is bound to a persistent data store. You can create relational database tables as a data store for rapid development by using the “Import DataSources” section in the SmartClient Admin Console. For deeper integration with your data tiers, see Chapter 8 (*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
<code>editRecord()</code>	starts editing an existing record
<code>editNewRecord()</code>	starts editing a new record
<code>saveData()</code>	saves the current edits (Add new records; Update existing records)

Databound components also provide several convenience methods for working with the selected records in a databound grid:

Convenience Method
<code>listGrid.removeSelectedData()</code>
<code>dynamicForm.editSelectedData(listGrid)</code>
<code>detailViewer.viewSelectedData(listGrid)</code>



`examples/databinding/component_databinding.jsp` shows most of these `DataSource` and databound component methods in action, with a `ListGrid`, `DynamicForm`, and `DetailView` that are dynamically bound to several different `DataSources`.



For more information, see the `DataSource` Operations, Databound Components, and Databound Component Methods subtopics under SmartClient Reference → Client Reference → Data Binding.

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 SmartClient components are:

5. **Create `DataSource` descriptors** (`.ds.xml` or `.js` files), specifying data model (metadata) properties in the `DataSource` fields.
6. **Back your `DataSources` with an actual data store.** The SmartClient Admin Console GUI creates and populates relational database tables for rapid development. Chapter 8 (*Data Integration*) describes the integration points for binding to production object models and data stores.
7. **Load `DataSource` descriptors** in your SmartClient-enabled pages with the `isomorphic:loadDS` tag (for XML descriptors in JSP pages) or client-only JS format. See *Creating DataSources* in the *SmartClient Reference* for more information.
8. **Create visual components** that support databinding (primarily form, grid, and detail viewer components).
9. **Bind visual components to `DataSources`** using the `dataSource` property and/or `setDataSource()` method.
10. **Modify component-specific presentation** properties in each databound component's `fields` array.

11. **Call databound component methods** (e.g. `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 8 ([Data Integration](#)) for an overview of server-side integration points that address all stages of your application lifecycle.

7. Layout

Component Layout

Most of the code snippets in this guide 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:

- consistently position your components relative to each other
- allocate available space based on relative measures (e.g. 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

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

The fundamental SmartClient 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 key properties of a layout manager are:

Layout property	Description
<code>members</code>	an array of components managed by this layout
<code>membersMargin</code>	number of pixels of space between each member of the layout
<code>layoutMargin</code>	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
<code>layoutAlign</code>	alignment with respect to the breadth axis of the layout ("left", "right", "top", "bottom", or "center")
<code>showResizeBar</code>	determines whether a drag-resize bar appears between this component and the next member in the layout (true false)
<code>width</code> or <code>height</code>	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`, `DetailView`, and `Layout` components have `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), `"scroll"` (always show scrollbars), or `"auto"` (show scrollbars only when needed).

You can specify layout members by reference, or by creating them in-line, and they may include other layout managers. 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:

```
isc.HLayout.create({
  ID: "myPageHeader",
  height: 50,
  layoutMargin: 10,
  members: [
    isc.Img.create({src: "myLogo.png"}),
    isc.LayoutSpacer.create({width: "**"}),
    isc.Label.create({contents: "My Title"})
  ]
})
```



See the SmartClient Demo Application (*SDK Explorer* → *Getting Started* → *Demo App*) for a good example of layouts in action



For more information, see SmartClient Reference → Client Reference → Layout.

Container Components

In addition to the basic layout managers, SmartClient 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 SmartClient Demo Application (*SDK Explorer* → *Getting Started* → *Demo App*) for examples of `SectionStack` and `TabSet` components in action.



For more information, see SmartClient Reference → Client Reference → 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 SmartClient *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
<code>numCols</code>	Total number of columns in the grid, for form controls and their titles. Set to a multiple of 2, to allow for titles, so <code>numCols:2</code> allows one form control per row, <code>numCols:4</code> allows two form controls per row, etc.
<code>titleWidth</code>	Number of pixels allocated to each title column in the layout.
<code>colWidths</code>	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
<code>colSpan</code>	number of form layout columns occupied by this control (not counting its title, which occupies another column)
<code>rowSpan</code>	number of form layout rows occupied by this control
<code>startRow</code>	whether this control should always start a new row (true false)
<code>endRow</code>	whether this control should always end its row (true false)
<code>showTitle</code>	whether this control should display its title (true false)

Field property	Description
<code>align</code>	horizontal alignment of this control within its area of the form layout grid ("left", "right", or "center")



See *Feature Explorer* → *Forms* → *Layout* for examples of usage of these properties

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

- `header`
- `blurb`
- `spacer`
- `rowSpacer`

To create one of these special controls, simply include a field definition whose `type` property is set to one of these four names. See the properties documented under `headerItem`, `blurbItem`, `spacerItem`, and `rowSpacerItem` for additional control.



For more information on form layout capabilities, see:

- *SmartClient Reference – Client Reference* → *Forms* → *DynamicForm*
- *SmartClient Reference – Client Reference* → *Forms* → *Form Items* → *FormItem*

8. Data Integration

SmartClient 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.

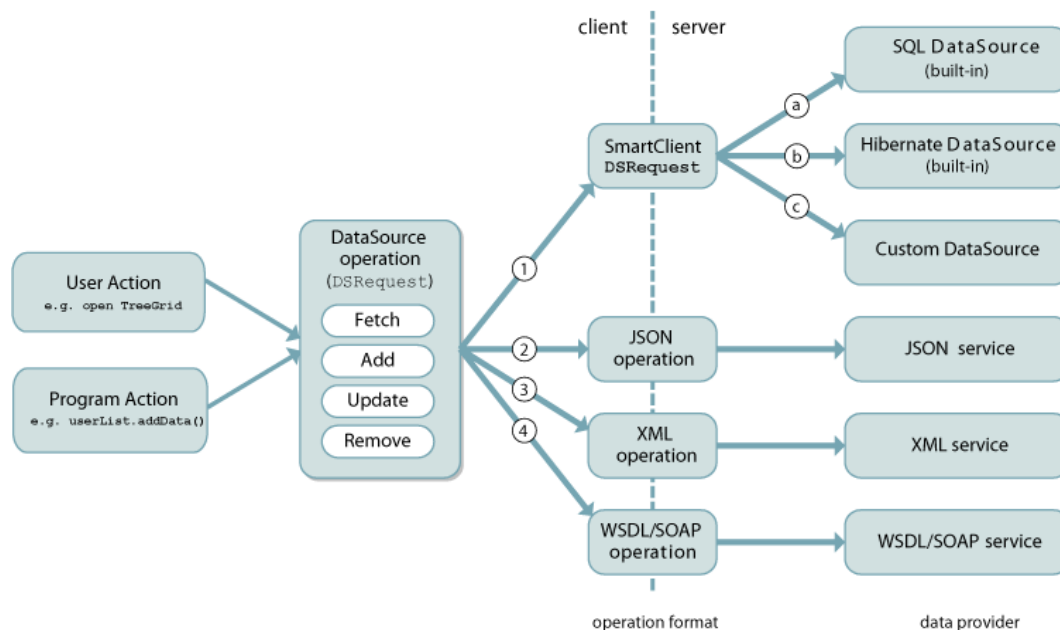
DataSource Requests

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 (SmartClient 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.



SmartClient Server Framework

Path 1 makes use of the SmartClient 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 SmartClient Server Framework for data integration. The server framework delivers an immense range of functionality that compliments any existing application and persistence engine. Chapter 9, [SmartClient Server Framework](#), provides an overview.

If you cannot use the SmartClient 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 SmartClient 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 9, [SmartClient 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:

- [SmartClient Reference – Client Reference > Data Binding > RestDataSource](#)
- [SmartClient Reference – Concepts > Client-Server Integration](#)



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

- SmartClient Showcase
http://www.smartclient.com/index.jsp#featured_restfulds

Criteria, Paging, Sorting and Caching

SmartClient 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, SmartClient does not *require* that a data provider implement all of these capabilities. In fact, SmartClient 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. SmartClient 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). SmartClient 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:

- [SmartClient Reference – Client Reference > Data Binding > ResultSet](#)
- [SmartClient Reference – Client Reference > Data Binding > DataSource.cacheAllData](#)

Authentication and Authorization

Securing SmartClient applications is done in substantially the same way as standard web applications. In fact, SmartClient'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 SmartClient 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 SmartClient, 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 `` 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 SmartClient 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 SmartClient 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 SmartClient, 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 JavaScript application logic files unprotected. If you are following SmartClient best practices, actual enforcement of security rules takes place on the server, so it doesn't matter if users can read the client-side code. However, if you are concerned that reading client-side code would help an attacker or competitor understand your application better, several free JavaScript obfuscators are available. As with other static resources, not protecting these files provides a performance boost.

If you are using the SmartClient 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 SmartClient, 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. SmartClient makes it easy to implement this ideal handling through the *Relogin* subsystem.

To enable SmartClient 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, SmartClient 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:

- [SmartClient Reference – Client Reference > RPC > Relogin](#)

Binding to XML and JSON Services

If you need to integrate with pre-existing XML or JSON web services (you are unable to install the SmartClient Server Framework *and* you are unable to use the `RestDataSource`), you can use SmartClient's support for XPath-based binding to XML or JSON responses.

To display XML or JSON data in a visual component such as a `ListGrid`, you bind the component to a `DataSource` which provides the URL of the service, as well as a declaration of how to form inputs to the service and how to interpret service responses as `DataSource` records.

An XPath expression, the `recordXPath`, is applied to the service's response to select the XML elements or JSON objects that should be interpreted as `DataSource` records. Then, for each field of the `DataSource`, an optional `valueXPath` can be declared which selects the value for the field from within each of the XML elements or JSON objects selected by the `recordXPath`. If no `valueXPath` is specified, the field name itself is taken as an XPath, which will select the same-named subelement or property from the record element or object.

For example, the following code defines a `DataSource` that a `ListGrid` could bind to in order to display an RSS 2.0 feed.

```
isc.DataSource.create({
  dataURL:feedURL,
  recordXPath:"//item",
  fields:[
    { name:"title" },
    { name:"link" },
    { name:"description" }
  ]
});
```

A representative slice of an RSS 2.0 feed follows:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<rss version="2.0">
<channel>
  <title>feed title</title>
  ...
  <item>
    <title>article title</title>
    <link>url of article</link>
    <description>
      article description
    </description>
  </item>
  <item>
    ...
```


Here, the `recordXPath` selects a list of `item` elements. Since the intended values for each `DataSource` field appear as a simple subelements of each `item` element (e.g. `description`), the field name is sufficient to select the correct values, and no explicit `valueXPath` needs to be specified.



For a running example of a ListGrid displaying an RSS feed, see *Feature Explorer → Data Integration → XML → RSS Feed*



For an example of using `valueXPath`, see *Feature Explorer → Data Integration → XML → XPath Binding*



For corresponding JSON examples, see *Feature Explorer → Data Integration → JSON → Simple JSON and JSON XPath Binding*

To retrieve an RSS feed, an empty request is sufficient. For contacting other kinds of services, the `dataProtocol` property allows you to customize how data is sent to the service:

Value	Description
"getParams"	Input data is encoded onto the <code>dataURL</code> , e.g. <code>http://service.com/search?keyword=foo</code>
"postParams"	Input data is sent via HTTP POST, exactly as an HTML form would submit them
"soap"	Input data is serialized as a SOAP message and POST'd to the <code>dataURL</code> (used with WSDL services)

Programmatic control of inputs and outputs is also provided.

`DataSource.transformRequest()` allows you to modify what data is sent to the service. `DataSource.transformResponse()` allows you to modify or augment the default `DSResponse` object that SmartClient assembles based on the `recordXPath` and `valueXPath` properties. This allows data transformations not possible with XPath alone, as well as integration of `DataSource` features such as data paging and validation errors with services that support those features.



For more information, see *SmartClient Reference – Client Reference → Data Binding → Client-side Data Integration*

WSDL Integration

SmartClient supports automated integration with WSDL-described web services. This support augments capabilities for integrating with generic XML services, and consists of:

- creation of SOAP XML messages from JavaScript application data, with automatic namespacing, and support for both "literal" and "encoded" SOAP messaging, and "document" and "rpc" WSDL-SOAP bindings
- automatic decode of SOAP XML messages to JavaScript objects, with types (e.g. an XML schema "date" type becomes a JavaScript `Date` object)
- import of XML Schema (contained in WSDL, or external), including translating XML Schema "restrictions" to SmartClient Validators

WSDL services can be contacted by using `XMLTools.loadWSDL()` or the `<isc:loadWSDL>` JSP tag to load the service definition, then invoking methods on the resulting `WebService` object.

`WebService.callOperation()` can be used to manually invoke operations for custom processing.



See Feature Explorer → Data Integration → XML → WSDL Web Services for an example of `callOperation()`

To bind a component to a web service operation, call

```
WebService.getFetchDS(operationName,elementName)
```

to obtain a `DataSource` which describes the structure of an XML element or XML Schema type named *elementName*, which appears in the response message for the operation named *operationName*. A component bound to this `DataSource` will show fields corresponding to the structure of the chosen XML element or type, that is, one field per subelement or attribute. `fetchData()` called on this `DataSource` (or on a component bound to it) will invoke the web service operation and load the named XML elements as data.

Similarly, `WebService.getInputDS(operationName)` returns a `DataSource` suitable for binding to a form that a user will fill out to provide inputs to a web service.

These methods allow very quick development, however, typically you cannot directly use the XML Schema embedded in a WSDL file to drive visual component `DataBinding` in your final application, because XML Schema lacks key metadata such as user-viewable titles.

You can create a `DataSource` that has manually declared fields **and** invokes a web service operation by setting `serviceNameSpace` to the `targetNamespace` of the `<definitions>` element from the WSDL file, and then setting `wsOperation` to the name of the web service operation to invoke. In this usage:

- creation of the operation input SOAP message is still handled automatically
- all of the custom binding facilities described in the preceding section are available, including XPath-based extraction of data, and programmatic manipulation of inbound and outbound data
- you can still leverage XML Schema `<simpleType>` definitions by setting `field.type` to the name of an XML Schema simple type embedded in the WSDL file.



See Feature Explorer → Data Integration → XML → Google SOAP Search for an example of these techniques.



The `targetNamespace` from the WSDL file is also available as `webService.targetNamespace` on a `WebService` instance.

For full read-write integration with a service that supports the basic `DataSource` operations on persistent data, `OperationBindings` can be declared for each `DataSource` operation, and the `wsOperation` property can be used to bind each `DataSource` operation (fetch, update, add, remove) to a corresponding web service operation.



To maximize performance, the *WSDL* tab in the Developer Console allows you to save a `.js` file representing a `WebService` object, which can then be loaded and cached like a normal JavaScript file.

9. SmartClient Server Framework

The SmartClient 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 SmartClient 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)
Long	Varies	sequence

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:



SmartClient Reference – Client Reference > Data Binding:

- `DataSource.schemaBean`
- `DataSource.inheritsFrom`
- `DataSource.autoDeriveSchema`



SmartClient Server JavaDoc:

- `com.isomorphic.datasources.DynamicDSGenerator`

Server Request Flow

When using the SmartClient 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 SmartClient 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:



SmartClient Reference – Concepts > Client-Server Integration

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:

SmartClient Reference – Client Reference > Data Binding > `DataSource.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 `SmartClient` 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:



SmartClient Reference – Client Reference > RPC > Direct Method Invocation

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 `SmartClient`, 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:



SmartClient Server JavaDoc:

- `com.isomorphic.datasource.DSRequest`
- `com.isomorphic.datasource.DSResponse`

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



SmartClient Reference:

- *Client Reference > RPC > RPCManager*
- *Client Reference > Forms > DynamicForm*

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



SmartClient Enterprise Showcase:

- <http://www.smartclient.com/index.jsp#userSpecificData>

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 deliver them to the browser as `Record` objects, ready for display by `DataBound` Components or processing by your code.

For example, if you are responding to a fetch, the following Java objects 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 `getPropertyName()` / `setPropertyName()` 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 SmartClient, 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`.

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



SmartClient 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:

- SmartClient 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:

SmartClient Reference – Client Reference > 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-SmartClient 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 SmartClient 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:

- SmartClient 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 `DataSource`s 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 `DataSource`s, it also limits the fields requested from the underlying data store. Custom `DataSource`s 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:

- *SmartClient Reference – Client Reference > Data Binding > 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 avoid 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 Declarative Security, see:

- SmartClient Reference:
 - Client Reference > Data Binding > OperationBinding.requiresRole*
 - Client Reference > Data Binding > DataSource.requiresAuthentication*
 - Client Reference > Data Binding > DataSourceField.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 declarative security, see:

- SmartClient 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**ppdate, **D**eleate pattern (called "add", "fetch", "update" and "remove" in SmartClient). SmartClient 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:

- *SmartClient Reference > Data Binding > DataSource.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 SmartClient 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"/>
    </validator>
  </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="$sessionId"/>
</operationBinding>
<operationBinding operationType="fetch">
  <criteria fieldName="sessionId" value="$sessionId"/>
</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">
  <serverObject className="com.sample.DMIHandler"
    methodName="doFullTextSearch"
    methodArguments="$criteria.accountId,false"/>
</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 SmartClient-specific server code at all*, without even the need to import SmartClient 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:

- SmartClient Reference:
Client Reference > Forms > Validator.serverCondition
- SmartClient Server Java Doc:
[com.isomorphic.datasource.DSRequest.addToTemplateContext\(\)](http://com.isomorphic.datasource.DSRequest.addToTemplateContext())
- Velocity User Guide (from the Apache foundation)
velocity.apache.org/user-guide

Server Scripting

SmartClient 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, including Java itself, as well as languages 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="java">
    String sessionId = session.getId();

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

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

Notice how even though the Java language is used, 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 Java 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
2. Scriptlets can easily be moved into normal .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 Java language even for teams that would not normally consider adopting a "scripting language".



For examples of Server Scripting see:

- SmartClient Enterprise Showcase:

http://www.smartclient.com/index.jsp#_Featured.Samples_Server.Examples_Server.Scripting



For more information on Server Scripting see:

- SmartClient Reference:

Concept> Server Script for SmartClient

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 `orderItems` in an order, they want to see the "itemName"s of the related `stockItems`, 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 `JPADDataSource` (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 SmartClient, 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 SmartClient. 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 SmartClient 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:

SmartClient Reference – Client Reference > Data Binding > DataSource > Custom Querying Overview

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:

- SmartClient Enterprise Showcase:
<http://www.smartclient.com/index.jsp#largeValueMapSQL>

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
<code>customValueFields</code>	Comma-separated list of fields to allow in <code>SELECT</code> clause despite being <code>customSQL="true"</code>
<code>customCriteriaFields</code>	Comma-separated list of fields to allow in <code>WHERE</code> clause despite being <code>customSQL="true"</code>
<code>excludeCriteriaFields</code>	Comma-separated list of fields to exclude from <code>\$defaultWhereClause</code>

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
<code>customSelectExpression</code>	Expression to use in <code>SELECT</code> and <code>WHERE</code> clauses
<code>customUpdateExpression</code>	Expression to use in <code>SET</code> clause of <code>UPDATE</code>
<code>customInsertExpression</code>	Expression to use in <code>VALUES</code> clause of <code>INSERT</code> . Defaults to <code>customUpdateExpression</code>

`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 SmartClient 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:

- SmartClient Reference – *Client Reference > Binding > DataSource:*

DataSourceField.customSQL

OperationBinding.customCriteriaFields

DataSourceField.customSelectExpression

DataSourceField.sqlStorageStrategy



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

- SmartClient Enterprise Showcase:
<http://www.smartclient.com/index.jsp#dynamicReporting>

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, 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 declaration 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 new functionality in a DataSource `.ds.xml` file, just reload the web page—the SmartClient 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, SmartClient 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 remote service to contact) or use them to control additional features you add to the built-in persistent `DataSources`.

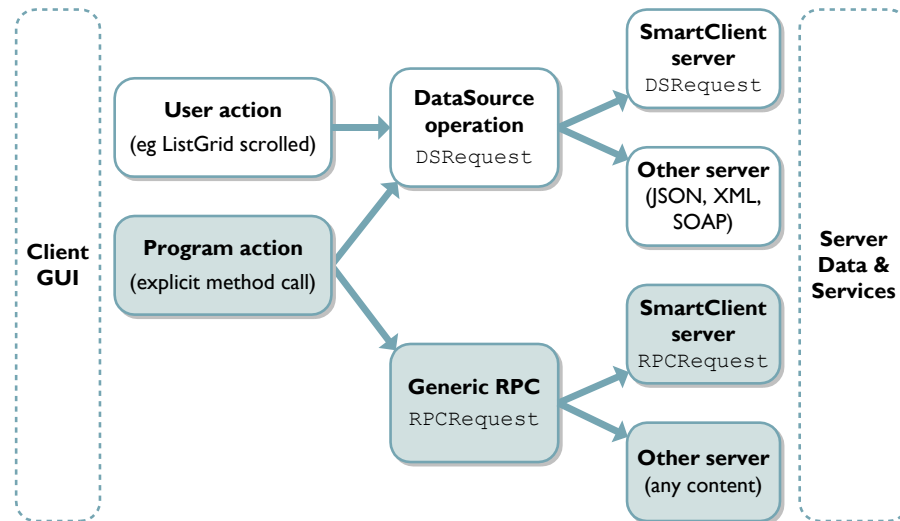


For more information on creating custom `DataSources`, see:

- SmartClient Reference - Concepts > Custom Server `DataSources`

Generic RPC operations (advanced)

Generic RPCs allow you to make arbitrary service calls and content requests against any type of server, but they also require you to implement your own request/response processing and GUI integration logic.



RPC operations sent to the SmartClient Java Server can use DMI declarations to route requests to appropriate server-side code, or a custom servlet can interact with the server-side `RPCManager` class to receive the `RPCRequest`.



For information about implementing RPCs with the SmartClient server, see the client and server documentation for `DMI`, `RPCManager`, `RPCRequest`, and `RPCResponse`:

- SmartClient Reference → Client Reference → RPC
- JavaDoc for `com.isomorphic.rpc`



`examples/server_integration/custom_operations/` shows how to implement, call, and respond to generic RPCs with the SmartClient Java Server

RPC operations can also be performed with non-SmartClient servers.

If you are using a WSDL-described web service, the operations of that web service can be invoked either through DataSource binding (as described under the heading *WSDL Integration* in the Data Integration chapter), **or** can be invoked directly via `webService.callOperation()`. Invoking `callOperation()` directly is much like an RPC operation, in that it allows you to bypass the DataSource layer and retrieve data for custom processing.

If you are not using a WSDL-described web service, you can retrieve the raw HTTP response from a server (in JavaScript String form) by setting the property `serverOutputAsString` on an `RPCRequest`. For an XML response, you may then wish to use the facilities of the `XMLTools` class, including the `parseXML` method, to process the response.

Responses that are valid JavaScript may be executed via the native JavaScript method `window.eval()`, or can be executed automatically as part of the RPC operation itself by setting `rpcRequest.evalResult`.



For information about implementing RPCs with non-SmartClient servers, see:

- *SmartClient Reference - Client Reference > RPC*
- *SmartClient Reference - Client Reference > Data Binding > Web Service (for WSDL-based RPCs)*

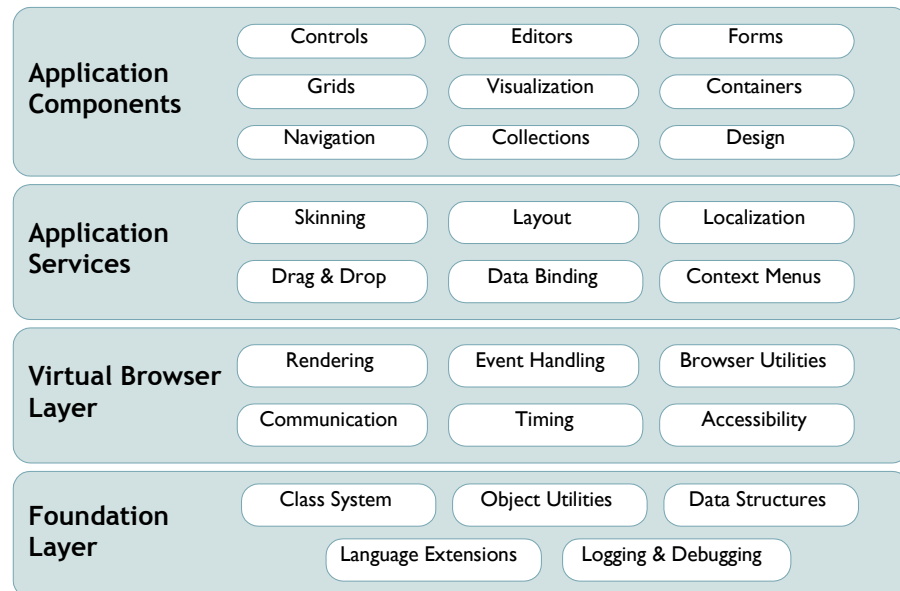
10. Extending SmartClient

SmartClient 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 SmartClient as *open* and *extensible* as possible.

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

Client-side architecture

The SmartClient client-side system implements multiple layers of services and components on top of standard web browsers:



From the bottom up:

- The **Foundation Layer** extends JavaScript to make it a viable programming language for enterprise applications. SmartClient adds true class-based inheritance, superclass calls, complex data structures, logging and debugging systems, and other extensions that uplift JavaScript from a lightweight scripting language, to a serious programming environment.
- The **Virtual Browser Layer** handles the most difficult part of rich web application programming—the vast collection of workarounds to avoid browser-specific bugs, and to implement consistent behavior across all supported browser types, versions, and modes. SmartClient makes web browsers *appear* to have standard rendering, event handling, communication, timing, and other behaviors—behaviors are not fully specified by web standards, or not implemented consistently in real web browsers.
- The **Application Services** layer provides higher level services that are shared by all SmartClient components and applications. This sharing radically reduces the footprint and complexity of rich web application code.
- The **Application Components** layer provides the pre-fabricated visual components—ranging from simple buttons, to interactive pivot tables—that you can assemble and data-bind to create rich web applications.

Earlier chapters of this guide have dealt primarily with the component layer—because most application development uses pre-fabricated components, most of the time. But all of these layers are open to you, and to third-party developers. If you need a new client-side feature, you can build or buy components that seamlessly extend SmartClient to your exact requirements. The following sections detail how.

Customized Themes

The first way to extend a SmartClient application is to change the overall look-and-feel of the 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.

The SmartClient SDK includes example themes (a.k.a. “skins”) for you to explore. Use Feature Explorer to browse through each theme.

You can specify a different user interface theme in the header of your SmartClient-enabled web pages:

- In the `isomorphic:loadISC` tag, set the `skin` attribute to the name of an available user interface skin, e.g.
`skin="SmartClient".`
- In a client-only header, change the path to `load_skin.js`, e.g.
`<SCRIPT SRC=../isomorphic/skins/SmartClient/
load_skin.js>`

The files for all available SmartClient user interface themes are located in the `/isomorphic/skins` directory. Each theme provides three collections of resources to specify look and feel:

Resource	Contains
<code>skin_styles.css</code>	a collection of CSS styles that are applied to parts of visual components in various states (e.g. <code>cellSelectedOver</code> for a selected cell in a grid with mouse-over highlighting)
<code>images/</code>	a collection of small images that are used as parts of visual components when CSS styling is not sufficient (e.g. <code>TreeGrid/folder_closed.gif</code>)
<code>load_skin.js</code>	component property overrides, to change default interactive behaviors (e.g. <code>listGrid.canResizeFields</code>) or high-level programmatic styling (e.g. <code>listGrid.alternateRecordStyles</code>)

You can customize component appearance in two ways:

1. **Create a custom skin:** to create a custom skin, copy an existing skin that most closely matches your intended skin and modify it. For example, let's say you wanted to customize the built-in "SmartClient" skin and call the resulting skin "BrushedMetal". The procedure is as follows:
 - a. Locate the "SmartClient" skin under `/isomorphic/skins` and copy the contents of that entire directory into a new folder called "BrushedMetal".
 - b. Edit the `/isomorphic/skins/BrushedMetal/load_skin.js` file. Find the line near the top of the file that reads:

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

and change it to:

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

- c. Delete the `/isomorphic/skins/BrushedMetal/load_skin.js.gz` and `/isomorphic/skins/BrushedMetal/skin_styles.css.gz` files.
 - d. Now you're ready to customize the new skin. You can do so by modifying any of the files listed in the table above 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.
 - e. Remember to change the name of the skin to the new skin name on your page to start using the new skin.
2. **Skin individual components:** set SmartClient component properties to use different styles, images, or behaviors. You can customize these properties on a per-class or per-instance basis.



For more information on Customized Themes, see:

- [SmartClient Reference - Concepts > Skinning/Theming](#)



See *Feature Explorer* → *Effects* → *Look & Feel* for examples of using skinning properties to customize component look & feel



The `load_skin.js` and `skin_styles.css` files for the SmartClient skin provide a good overview of available skinning properties. Individual properties can be looked up in the *SmartClient Reference*.

Customized Components

The easiest way to extend the SmartClient component set is to subclass and customize existing components.

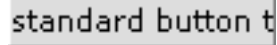
The two essential methods for customizing SmartClient component classes are:

```
isc.defineClass(newClassName, baseClassName)
isc.newClassName.addProperties(properties)
```

For example, let's say you want a customized button component that draws bigger, bolder buttons. The standard SmartClient `Button` component has a size of 100 by 20 pixels, a non-wrapping title, and styling based on CSS style names that begin with "button." So this code:

```
isc.Button.create({title:"standard button title"});
```

will create a component that looks like this:



To create and customize a subclass of the standard `Button`, you could define a `BigButton` class as follows:

```
isc.defineClass("BigButton", Button);
```

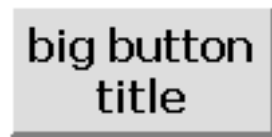
and add/override relevant properties on this class as follows:

```
isc.BigButton.addProperties({
  height:50,
  overflow:"visible",
  baseStyle:"bigButton",
  wrap:true
});
```

Now the following code:

```
isc.BigButton.create({title:"big button title"});
```

will create components that look like this:




`examples/custom_components/BigButton` contains the code for this example (including the “bigButton” CSS style definition).

New Components

If you need to extend beyond the customizable properties of the standard SmartClient 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`.

Again, you can use `defineClass()` to define a new class, e.g.

```
isc.defineClass("myWidget", Canvas)
```

In addition to instance properties, new components typically add instance methods, and may also add class (i.e. static) properties and methods. The core interfaces to flesh out a new component class are:

```
className.addProperties(properties)
className.addMethods(methods)
className.addClassProperties(properties)
className.addClassMethods(methods)
```



For more information on these and other class-creation interfaces, see “Class” and “ClassFactory” under *SmartClient Reference* → *Client Reference* → *System*.



`examples/custom_components/` contains the source code for several visual components—including `SimpleLabel`, `SimpleSlider`, and `SimpleHeader`—that are referenced below. These examples are your best starting points for building new SmartClient components.



Before you begin development of an entirely new component, try the SmartClient 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.

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

1. Create a **Layout** subclass that generates and manages a set of other components.

This approach is demonstrated in the `SimpleHeader` example, which automatically generates member components for the header image, spacer, and title. This is a fairly trivial example; Layout subclasses are more often used 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.

1. Create a **Canvas** subclass that generates and configures a set of other foundation components.

This approach is demonstrated in the `SimpleSlider` example, which builds an interactive slider widget out of a `Canvas` parent, `StretchImg` track element, and `Img` thumb element. The SmartClient foundation components entirely buffer this code from browser-specific interpretations of HTML, CSS, events, etc.

2. Create a **Canvas** subclass that contains your own HTML and CSS template code.

This approach is demonstrated in the `SimpleLabel` example. It provides the most flexibility to create components using any feature of HTML and CSS. However, it also requires that you test, optimize, and maintain your code on all supported web browsers. Whenever possible, you should use SmartClient foundation components instead to buffer your code from browser inconsistencies.



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

SmartClient Reference - Concepts > DOM Integration & Third-party Components



Whenever you add new properties or methods to a SmartClient class or subclass, you should name them with a unique prefix, to avoid future naming conflicts with other interfaces. If you intend to deploy your extensions in portals or other environments where interoperability is a concern, Isomorphic can confirm and reserve a namespace for your interfaces. Please contact namespaces@smartclient.com for assistance.

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.icons`).
3. Build a custom picker based on any standard or custom SmartClient components and services (see above).
4. Respond to end-user click events on that icon to show your picker (see `FormItemIcon.click`) 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.



`examples/custom_components/CustomPicker` contains example code for `YesNoMaybeItem`, a form control that displays a custom picker with Yes, No, and Maybe buttons. This example also demonstrates the use of static (class) methods and properties in SmartClient components.

New form controls can alternatively be implemented via the `CanvasItem` class, which allows any SmartClient 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 `FormItem`s 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, so it does not cause the `ListGrid` row to expand when editors are shown. 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 see:

- [*SmartClient Reference – Client Reference > Forms > Form Items > CanvasItem*](#)

SmartClient Enterprise Showcase:

- [*http://www.smartclient.com/index.jsp#nestedEditor*](http://www.smartclient.com/index.jsp#nestedEditor)

11. Tips

Beginner Tips

- **Pay extra attention to commas in your JavaScript code.**

Specifically in JavaScript Object literals, like the properties passed to `create()`. Missing commas between properties, or an extra comma after the last property, are among the most common syntax errors.

- **Use the Developer Console for dynamic testing.**

SmartClient eliminates the need to instrument your JS code for quick tests. Simply open the Developer Console to inspect and interact with components on-the-fly. The JS evaluator provides a quick means to make direct method calls while your application is running.

- **Use SmartClient logging to debug your applications.**

At minimum, use `Log.logWarn()` to log debugging messages in the background, instead of `alert()` calls that disrupt user experience and application flow. For even more control, you can take advantage of log scoping, priorities, and conditionals. See *SmartClient Reference – Concepts → Debugging*

HTML and CSS Tips

- **Use SmartClient 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. SmartClient 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. SmartClient'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 SmartClient components only through their published APIs.**

SmartClient 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 constantly improved and optimized in new releases of SmartClient. The only stable, supported way to manipulate a SmartClient 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.

Architecture Tips

- **Leverage the SmartClient Ajax architecture for optimal performance, responsiveness, and scalability.**

The classic web application model, in which a new page is rendered on the server for every client request, is very inefficient. With SmartClient components and services, your web applications

can make background data and service requests while users continue to interact with the front-end GUI. This “Asynchronous JavaScript and XML” (Ajax) model can radically improve usability and performance across the board, or specifically in your most critical workflows.

In brief: *Move the presentation workload to the client.* The SmartClient client-side engine handles:

- complex HTML rendering
- component layout
- view navigation
- read-only operations (filter, sort, find, etc) on cached data

So user interruptions can be virtually eliminated, and server round-trips minimized to those required for data/service calls and secure business logic.

- **Structure your code for optimal client caching.**

Since SmartClient provides client-side component rendering and page layout, it is possible to cache most of the structure and logic of your presentation on the client, for even better performance. Specifically: *Avoid server-side templating of SmartClient JS or SmartClient XML code files.* Your goal should be a bootstrap page with a block of templated JS variables, followed by a set of static, cacheable JS or XML includes. Those included files will contain either:

- declarative SmartClient UI and DataSource descriptors, or
- client-side logic that references the initial dynamic/templated variables from the bootstrap page, as well as dynamic properties and data fetched via RPCs after the page has loaded

For web applications that are deployed over slow WAN, dial-up, or cellular links, you may want to integrate the optional *Network Performance* module. This SmartClient module provides explicit caching control, as well as server-side file packaging and compression services, for optimal performance on slow networks.

- **Load many components at once, and defer creating/drawing each component until it must be shown to the end user.**

The average SmartClient component definition is 10 to 50 times smaller than the corresponding static HTML. You can therefore load hundreds of visual components, representing dozens of

unique application views, in the time and memory that are normally used for a single HTML page.

However, it does take time and memory to create and draw all of those components on the client. For immediate responsiveness, you will want to create and draw only the components required for the initial view. Other pre-loaded components may be created and drawn on-the-fly.

- To defer creating a component, wrap the `create()` call in a JS function that you can call on demand. If you take this approach, you can also `destroy()` components to free up client resources, and later re-create them from your constructor function.
- To defer drawing a component, set its `autoDraw` property to `false`. Or call the global `isc.setAutoDraw(false)` to disable automatic drawing for all subsequently created components. To explicitly draw a component, call `draw()`. You can also `clear()` components to free up client resources, and call `draw()` again 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.

For more information on architecting your applications for high-performance, client-side view navigation, see *SmartClient Reference* → *Concepts* → *SmartClient Architecture*.

12. Evaluating SmartClient

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

Which Edition to Evaluate

SmartClient 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 SmartClient includes a sample project with a pre-configured Hypersonic SQL Database, which you can use to evaluate all of the capabilities of SmartClient'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, SmartClient'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, SmartClient 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 SmartClient 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 SmartClient'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 SmartClient 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 SmartClient.

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

Evaluating Performance

SmartClient 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 are 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 SmartClient. If you have followed SmartClient best practices, your 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:

- 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.

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 SmartClient 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 SmartClient 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 SmartClient 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, please feel free to contact the SmartClient product team:

<i>Web</i>	smartclient.com
<i>General</i>	info@smartclient.com feedback@smartclient.com
<i>Evaluation Support</i>	forums.smartclient.com
<i>Licensing</i>	sales@smartclient.com

We welcome your feedback, and thank you for choosing SmartClient.

End of Guide