Isomorphic
SOFTWARE

SmartClient™ Quick Start Guide

SmartClient™ Quick Start Guide
SmartClient v6.5

Isomorphic Software, Inc.
109 Stevenson Street, Level 4
San Francisco, CA 94105
U.S.A.

Web: **www.isomorphic.com**
Email: **info@isomorphic.com**

Contents

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 sections, each presenting a set of concepts and hands-on information that you will need to build SmartClient-enabled web applications. These sections are intended to be read in sequence—earlier sections provide the foundation concepts and configuration for later sections.

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* (Section 2) to start a SmartClient development server and begin *Exploring* (Section 3) and *Coding* (Section 4). But if you can spare a few minutes, we recommend reading the introductory sections first, for the bigger picture of SmartClient goals and architecture.

Thank you for choosing SmartClient, and welcome.

Why SmartClient?

If you are reading this guide, you may have already received sufficient proof of SmartClient's value to you and your work. But to keep our marketing people happy, here is a summary of that value:

> *SmartClient 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.*

What distinguishes SmartClient from other web presentation layers is that it compromises neither application usability, nor zero-install web deployment. SmartClient Ajax (Asynchronous JavaScript & XML) applications are simultaneously *rich client* applications, providing high-productivity interfaces to end users, and *thin client* applications, running in the standard web browsers available on every workstation, desktop, and laptop today. They are also *open* applications, providing total extensibility, interoperability with existing systems, and integration with existing code.

Rich Client

Simply put, SmartClient brings user interface intelligence back to the client, for faster, more intuitive applications. Today's client systems—machines often clocked in gigahertz—are left idling while application servers render and return the most basic HTML pages. SmartClient harnesses that power by performing complex user interface *rendering*, *navigation*, and *data operations* on the client.

For ***developers***, the result is a more logically structured, simplified, maintainable approach to web front-end development. SmartClient supports a natural separation of user interface logic and business logic between client and server, so you are not forced to implement complex artificial boundaries separating these layers in your server-side code. However, SmartClient is fully compatible with common Model-View-Controller (MVC) frameworks like Apache Struts or Tapestry, so you may continue to leverage your existing systems and skills.

For ***end users***, the result is a more productive, responsive, flicker-free application. Client and server no longer need to communicate for every single action in the user interface. When they do communicate, it is by transparent remote procedure calls (RPCs), carrying the minimum of required operations and data.

For ***businesses***, the result is a more scalable, efficient production system. Every new user of a SmartClient application effectively brings their own CPU to handle user interface rendering and state management. Application servers are freed to handle the jobs for which they were originally intended: executing secure business logic, and brokering data and services from other tiers.

Thin Client

The defining trait of a ***web application*** is that it deploys to standard web browsers. Any client program may access a web application server, but only a browser-based client will run without software installation on every desktop and laptop running in offices and homes today. In an increasingly connected world, client-side installation is no longer an option for most applications.

SmartClient is the only rich-client system to maintain the zero-install deployment model that makes web applications so appealing, despite the shortcomings of web standards for building applications.

To achieve this, SmartClient implements multiple layers of services and components based upon those standards. These rich capabilities are then optimized and tested on every popular web browser and operating system. Essentially, SmartClient treats browser Document Object Model (DOM) calls as the new "assembly language", and HTML as the new "pixels". So SmartClient applications run without software download, installation or configuration, on standard web browsers including Internet Explorer, Firefox, Mozilla, Netscape, Safari and Opera, on Windows, Linux, Solaris, and MacOS operating systems.

Open Architecture

SmartClient combines the best aspects of rich client and thin client—but that's just the beginning of the story.

First, because SmartClient is built entirely with ***standard*** web technologies, it integrates perfectly with your existing web content, applications, portals, and portlets. You can upgrade existing web applications and portals at your own pace, by weaving selected SmartClient components and services into your HTML pages. You can reuse existing content and portlets by embedding them in SmartClient

user interface components. SmartClient allows a smooth *evolution* of your existing web applications—you don't have to start over.

Next, SmartClient is fully ***open*** to integration with other technologies. On the client, you can seamlessly integrate ActiveX controls, Java applets, Flash/Flex modules, Scalable Vector Graphics (SVG), and other client technologies for visualization, streaming media, desktop application 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 totally ***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.

Proven Technology

In a nutshell: SmartClient provides the high-performance, rich-GUI architecture of traditional ***client-server*** applications, but implemented with the standard technologies of ***web*** applications (HTML, CSS, JavaScript, XML, HTTP). And it does so without limiting your flexibility, and without requiring a major redesign of your existing applications.

Isomorphic Software focuses exclusively on SmartClient products and services. We supply and support the toughest customers in the industry—other software developers—ranging from cutting-edge startups, to the largest enterprise application vendors.

As a result, SmartClient is the most heavily used, broadly proven rich-client web presentation layer available today. Hundreds of thousands of end users work with SmartClient-enabled applications, in systems including:

- CRM and customer support
- intranet portal servers
- financial analysis
- document management
- supply chain extranets
- business intelligence dashboards
- military command and control
- consumer applications

SmartClient is a living, growing system, continuously improved by Isomorphic, our customers, and their customers. What can we do for you? 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.
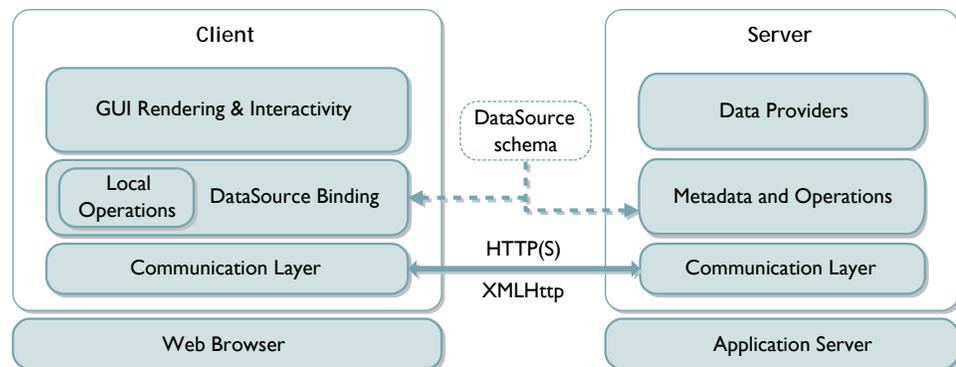
At the client web browser, SmartClient provides a deep stack of services and components for rich Ajax (Asynchronous JavaScript and XML) applications.

With or without the optional SmartClient Java Server, SmartClient provides a cohesive client-server data binding architecture. SmartClient's model-driven components understand metadata formats such as XML Schema, allowing you to create a rich data model that is shared across client and server, including high-level interfaces for the most common data operations (Add-Fetch-Update-Remove; aka Create-Retrieve-Update-Delete).

However, 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.

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 `Canvas`, `Img`, `StretchImg`, and `StatefulCanvas`.
Event Handling	SmartClient event handling systems, including mouse, keyboard, focus, drag & drop, enable/disable, and selection capabilities.
Controls	Basic visual controls, including `Button`, `Toolbar`, `Menu`, and `Menubar`.
Forms	Form layout managers, value managers, and controls (including `TextItem`, `DateItem`, `CheckboxItem`, `SelectItem`, etc.).
Grids	`GridRenderer`, `ListGrid` and related subclasses, providing grid rendering, selection, sorting, editing, column handling, and cell events.
Trees	`Tree` data structures, and `TreeGrid` UI components, for managing hierarchical data.
Layout	Component layout managers and layout-managed containers, including `HLayout`, `VLayout`, `Window`, and `TabSet`.
Data Binding	Data model, cache management, and communication components including `DataSource`, `ResultSet`, and `RPCManager`.
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 `CubeGrid` 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.
Client Bridges	Client-side components and services for deep integration (including layering, drag-and-drop, and bi-directional communication) with: • Java applets • ActiveX controls • Flash/Flex plug-ins • SVG (Scalable Vector Graphic) documents

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 evaluation, prototyping, and 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 prototyping. 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 rapid prototyping and 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.

Section 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: JDK 1.4 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:
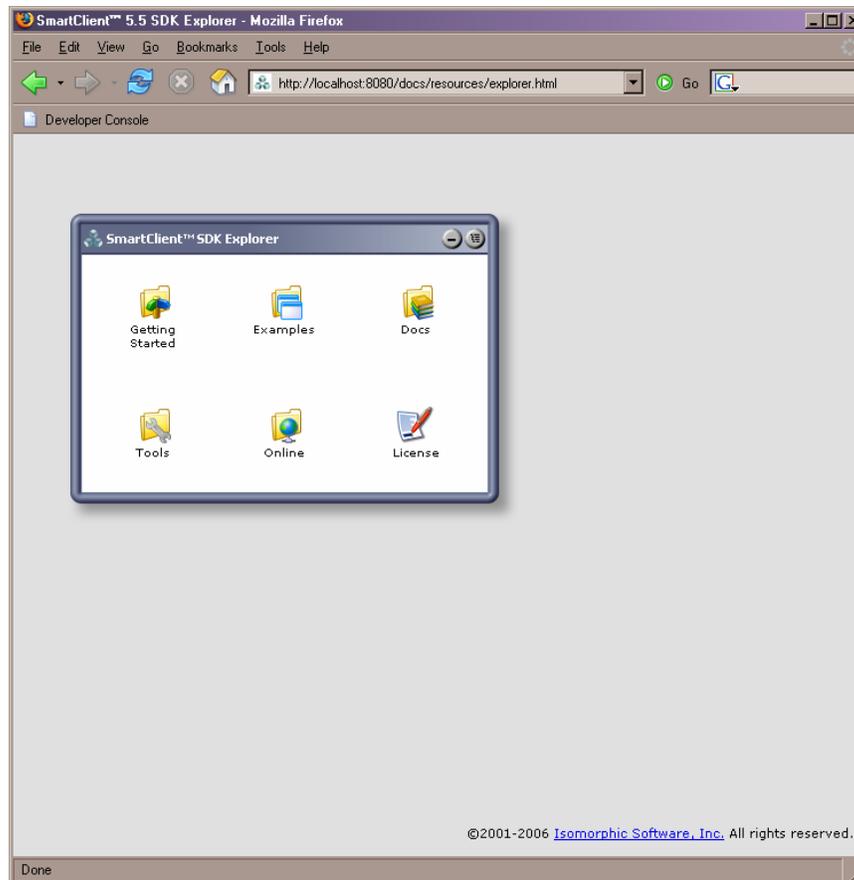
Browser Configuration (recommended)

If you are using a recent version of Internet Explorer, you may need to enable your browser to display interactive web pages directly from the file system. This approach is useful for stand-alone examples and test cases.

Go to Internet Options... > Advanced > Security, and select "Allow active content to run in files on My Computer" to enable JavaScript in local files. This will allow you to run SmartClient-enabled HTML pages simply by opening them (e.g. double-clicking) from your file system.

This step is not required in Mozilla/Firefox web browsers.

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. Exploring

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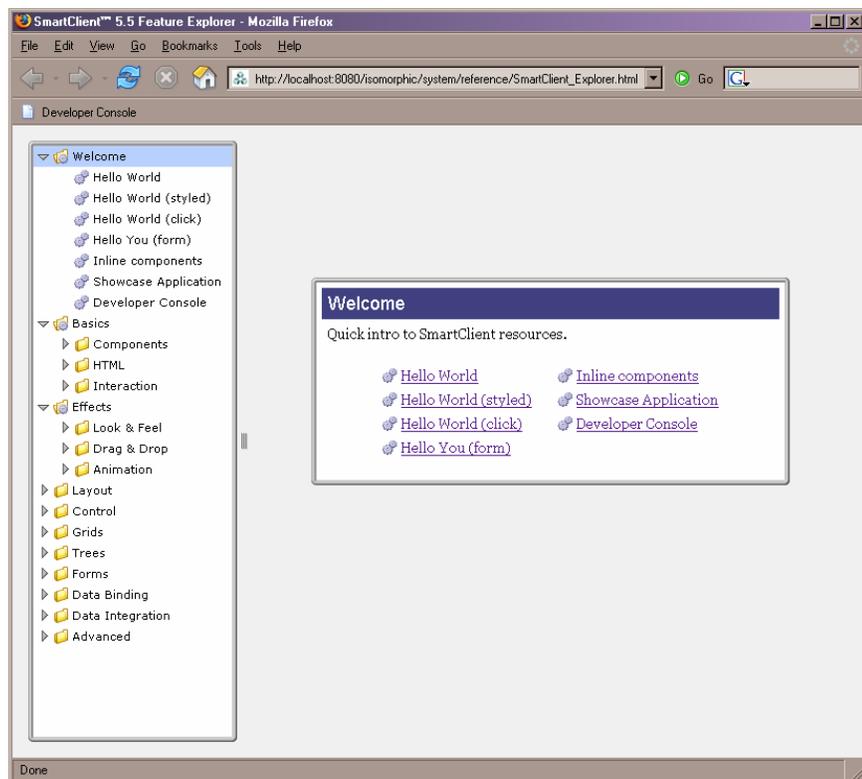
The Feature Explorer is your best starting point for exploring SmartClient capabilities and code.

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 the *Headers* section 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.

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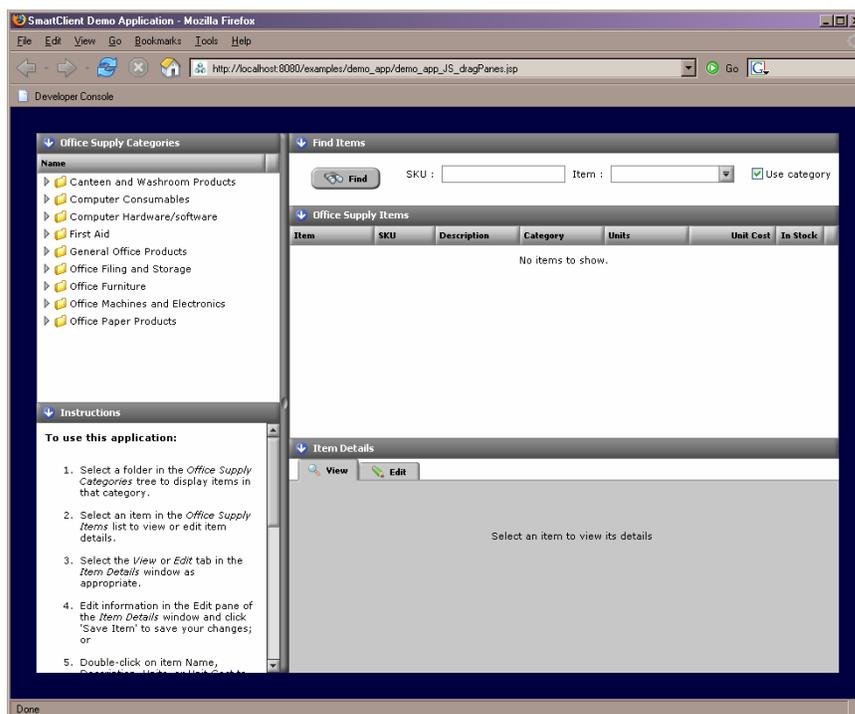
This example application demonstrates a broad range of SmartClient user interface, data binding, and layout features.

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 Section 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 sections 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.

SmartClient 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:
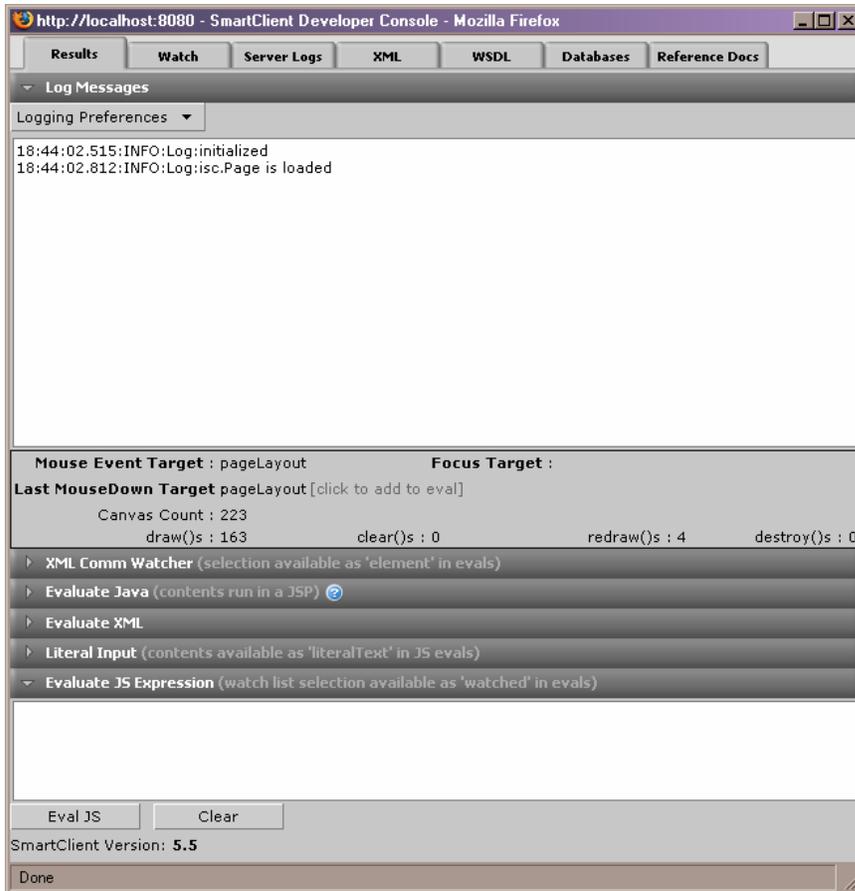
⚠ **Popup blocker utilities may prevent the Developer Console from appearing.** If this happens, you must instruct your popup blocker to allow this window. Please refer to the documentation for your specific browser or blocker utility. Holding the **Ctrl** key while opening the console will allow the popup in most systems.

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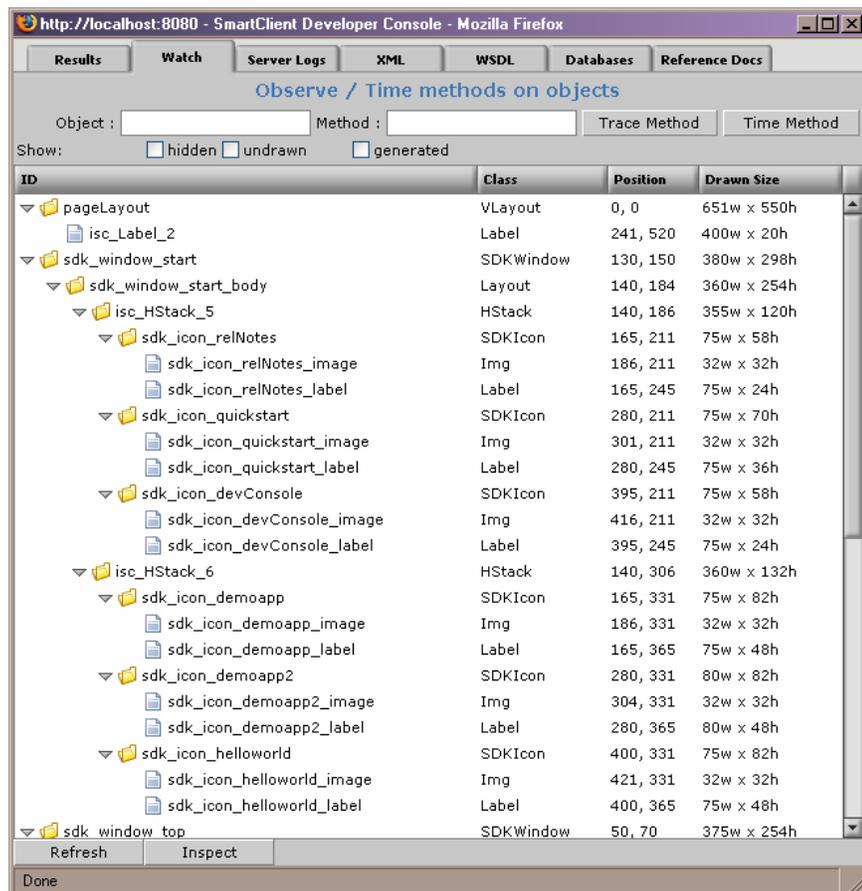
In the **Watch** pane, you may:

- 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. 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.

SmartClient 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. Ener 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.

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. e.g., `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. e.g. `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, e.g.

```
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 attibutes:

```
<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 (e.g. by double-clicking the file's icon), provided your browser allows interactive web pages to run from your file system (see "Browser Configuration", page 7).

You must open the `.jsp` versions through your server, e.g.

```
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.

The next section 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 section provides basic usage information for the independent components only. Managed form controls are discussed in more detail in the next two sections of this guide.

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 *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 section describes basic management and manipulation of ***independent visual components*** only. For information on the creation and layout of managed form controls, see Sections 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 (e.g. `button` or `btn`)
- include the component's action (e.g. `update`)
- include the datasource the component affects (e.g. `salesOrder`) e.g., `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. Section 7 (*Layout*) introduces the SmartClient Layout managers, which you can use to automatically size, position, and reflow your components at runtime.

Hiding & 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)	✔	✔	
`DetailViewer`	✔		

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 section outlines the *client-side* interfaces that you may use to configure databound components and interact with their underlying datasources. Section 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 (e.g. title, width, alignment). Component fields are discussed immediately below.

- ***DataSource*** fields provide ***metadata*** describing the objects in a particular datasource (e.g. data type, length, required). DataSource fields are discussed under "DataSources" later in this section.

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)
DetailViewer	rows

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 DetailViewer, 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" later in this section 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:

> 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 under
"DataSources" later in this section.

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:

For more information on the *layout* of managed form controls, see "Form Layout" in Section 7 (*Layout*).

DataSources

SmartClient *DataSource* objects provide a presentation-independent, implementation-independent description of a set of persistent data fields. DataSources 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 (e.g. `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 DataSources*). 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 exactly one field with `primaryKey="true"`. The primaryKey field must have a unique value in each data object (record) in a DataSource. A primaryKey field is not required for read-only DataSources, but it is a good general practice to allow for future Add, Update, or Remove data operations.

Following these rules, a DataSource descriptor for the "contacts" example earlier in this section 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 definitons 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"` | `"sequence"`
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`; whether this is the field that uniquely identifies each record in this DataSource (i.e., it must have a unique value for each record). Each DataSource must have exactly one field with `primaryKey="true"`. The primaryKey field is often specified with `type="sequence"` and `hidden="true"`, to generate a unique internal key for rapid prototyping.
foreignKey	a reference to a field in another DataSource (i.e., `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	`fetchData(…)`	retrieves records from the datasource that exactly match the provided criteria
	`filterData(…)`	retrieves records from the datasource that contain (substring match) the provided criteria
Add	`addData(…)`	creates a new record in the datasource with the provided values
Update	`updateData(…)`	updates a record in the datasource with the provided values
Remove	`removeData(…)`	deletes a record from the datasource that exactly matches the provided criteria

These methods each take three parameters:

- a **data** object containing the criteria for a Fetch or Filter operation, or the values for an Add, Update, or Remove operation

- a **callback** expression that will be evaluated when the operation has completed

- a **properties** object containing additional parameters for the operation—timeout length, modal prompt text, 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 prototyping by using the "Import DataSources" section in the SmartClient Admin Console. For deeper integration with your data tiers, see Section 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
`editRecord()`	starts editing an existing record
`editNewRecord()`	starts editing a new record
`saveData()`	saves the current edits (Add new records; Update existing records)

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

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

    `examples/databinding/component_databinding.jsp`
shows most of these DataSource and databound component
methods in action, with a ListGrid, DynamicForm, and DetailViewer
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 section began by introducing Databound Components, to build on
the concepts of the previous section (Visual Components). However, in
actual development, DataSources usually come first. The typical steps to
build a databound user interface with SmartClient components are:

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

2. **Back your DataSources with an actual data store.** The
   SmartClient Admin Console GUI creates and populates relational
   database tables for rapid prototyping. Section 8 (*Data
   Integration*) describes the integration points for binding to
   production object models and data stores.

3. **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.

4. **Create visual components** that support databinding (primarily
   form, grid, and detail viewer components).

5. **Bind visual components to DataSources** using the
   `dataSource` property and/or `setDataSource()` method.

6. **Modify component-specific presentation** properties in each databound component's `fields` array.

7. **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 Section 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
`members`	an array of components managed by this layout
`membersMargin`	number of pixels of space between each member of the layout
`layoutMargin`	number of pixels of space surrounding the entire layout

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

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

⚠️ **Components that automatically size to fit their contents will not be resized by a layout manager.** By default, `Canvas`, `Label`, `DynamicForm`, `DetailViewer`, and `Layout` components have `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
`numCols`	Total number of columns in the grid, for form controls and their titles. Set to a multiple of 2, to allow for titles, so `numCols:2` allows one form control per row, `numCols:4` allows two form controls per row, etc.
`titleWidth`	Number of pixels allocated to each title column in the layout.
`colWidths`	Optional array of pixel widths for all columns in the form. If specified, these widths will override the column widths calculated by the form layout manager.

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

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

align	horizontal alignment of this control within its area of the form layout grid (`"left"`, `"right"`, or `"center"`)

See *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

Like client-server desktop applications, SmartClient browser-based applications interact with remote data and services via background communication channels. Background requests retrieve chunks of data rather than new HTML pages, and update your visual components in place rather than rebuilding the entire user interface.

SmartClient supports two general classes of client-server operations:

DataSource operations.....standard Fetch, Add, Update, and Remove operations on structured data, with built-in, automatic GUI component behaviors

RPC operations....................general-purpose RPCs (Remote Procedure Calls), requiring custom request/response handling and GUI integration code

This section focuses primarily on data integration for the *DataSource Operations*, which were introduced in Section 6 (*Data Binding*) of this guide.

A visual component which has been bound to a DataSource will originate DataSource Operations, either automatically in response to user actions, or manually in response to calls to the DataBound component methods. Data Integration is the process of satisfying those requests by adapting SmartClient to your existing servers or to third-party services you wish to consume.

You can integrate with any DataSource operation in either of two places:

Server-side .... using SmartClient server components (on any supported Java application server) for rapid prototyping against database tables, or for integration with existing Java business logic and data models

Client-side............. using the SmartClient request/response transformation pipeline to integrate with JSON or XML data providers (including WSDL-described web services)

The following diagram shows the available server-side and client-side data integration paths in more detail:
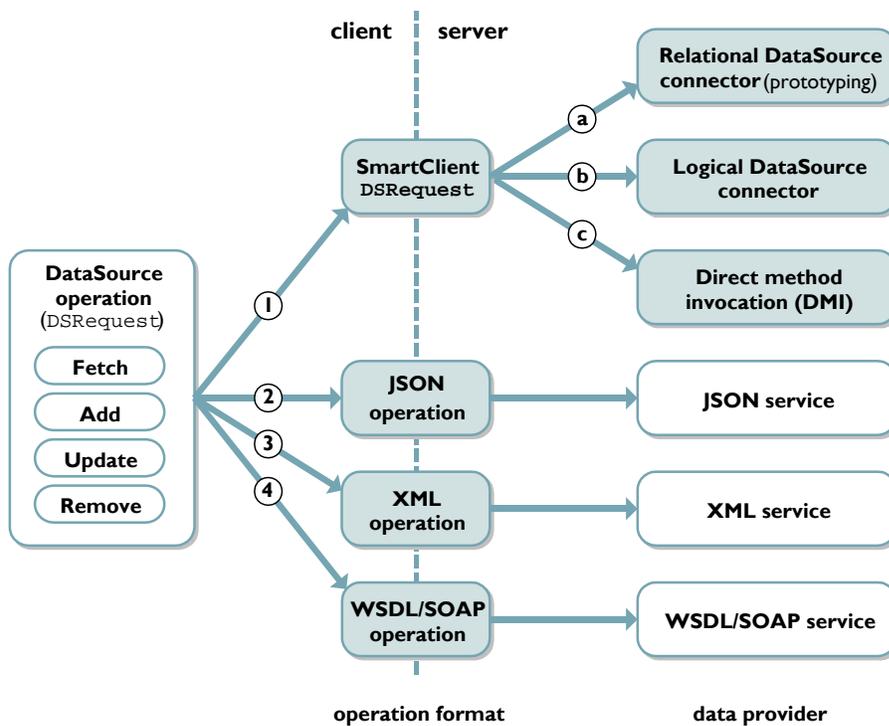
These paths roughly correspond to the following types or levels of application development:

Development Type	Paths
Rapid Prototyping (SmartClient Server)	1a
Java Server Integration (SmartClient Server)	1b, 1c
SOA (Service Oriented Architecture) Integration	2, 3, 4

For a large application, you may choose to follow different integration paths at different stages of development. For example, you could start prototyping an application immediately with the Relational DataSource connector (1a), move to custom DataSource connectors (1b) or DMI (1c) as your business logic is finalized, and move to XML (3) or SOAP (4) integration when your business logic provides stable service interfaces.

You may also use any or all of these paths in parallel within the same application. For example, you could use Java server integration (1b, 1c) for your internal business logic, JSON service integration (2) to integrate an external search engine like Yahoo!, and WSDL/SOAP integration (4) to integrate hosted enterprise services like salesforce.com.

SmartClient provides this range of integration options so you can choose the best approach for the job at hand. These approaches are discussed in more detail in the following sections.

Rapid Prototyping (path 1a)

You can use pre-fabricated DataSource connectors to bind directly to a SQL data store. This is the fastest way to get started with SmartClient development. You can implement many common interactions—and many complete, simple applications—without writing any server-side logic. Note that this path requires the SmartClient Server.

The SmartClient SDK includes an embedded HSQL database, pre-populated with sample datasets, and a built-in object-relational (OR) connector for rapid prototyping using the embedded HSQL database or using external DB2, Oracle, SQL Server, MySQL, and PostgreSQL databases.

The SmartClient Admin Console provides a browser-based GUI to configure these database connections, generate database tables from your DataSource descriptors, and populate those tables with test data. Click the **Tools** > **Admin Console** link in the SDK Explorer, or browse directly to http://localhost:8080/tools/adminConsole.jsp, to open the console.

> For running examples that use the built-in OR connector, see the *Data Binding* section of the *Feature Explorer*, and also *SDK Explorer > Examples > Data Binding*

> For more information, see:
>
> - *SmartClient Reference – Client Reference > Data Binding > DataSource > SQL DataSources*
>
> - *SmartClient Reference – Client Reference > Data Binding > DataSource > Admin Console*

Java Server Integration (paths 1b, 1c)

In this approach, DataSource requests issued by DataBound components arrive on the server as Java Objects. You deliver responses to the browser by returning Java Objects.   Note that this path requires the SmartClient Server.

There are two approaches for routing inbound requests to your business logic:

> ***RPCManager dispatch***......... inbound requests are handled by a Java servlet or .jsp that you provide. The `RPCManager` is used to retrieve requests and provide responses

> ***Direct Method Invocation***....XML declarations route requests to existing business logic methods. Inbound request data is adapted to method parameters, and method return values are delivered as responses

Which approach you use is largely a matter of preference. Direct Method Invocation (DMI) may allow simple integration without writing any SmartClient-specific server code. RPCManager dispatch integration provides an earlier point of control, allowing logic that applies across different DataSource operations to be shared more easily.

> For more information, see:
>
> * *SmartClient Reference - Java Server Reference > Server DataSource Integration*

Service-Oriented Architecture (paths 2, 3, 4)

SmartClient supports declarative, XPath-based binding of visual components to web services that return 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 (eg `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 `dataURL`, eg http://service.com/search?keyword=foo
"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 `dataURL` (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 (eg 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 prototyping, 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 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.

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.
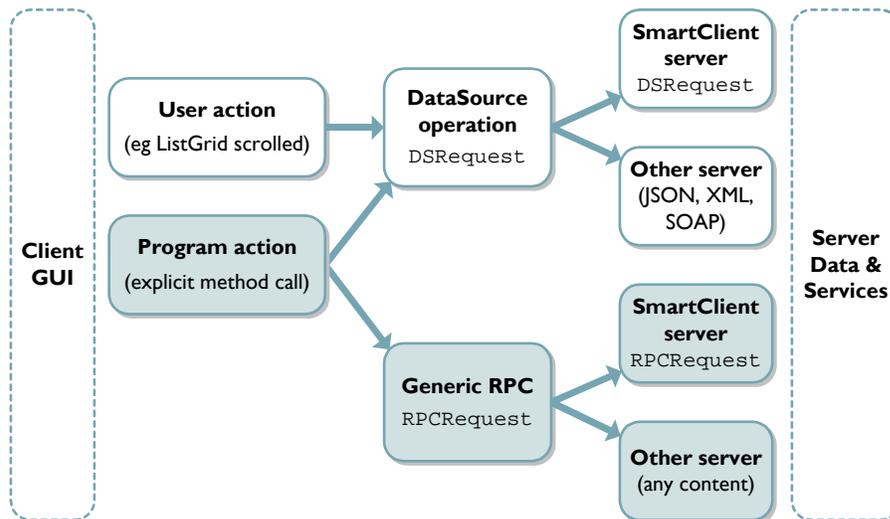
As with DataSource operations, RPC operations sent to the SmartClient Java Server can use two methods to route requests to appropriate server-side code: Direct Method Invocation (DMI) or RPCManager dispatch.

> 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 this 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. However, unlike a normal RPC, the web service definition provides a schema for the inputs and outputs of the operation.

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 `isc.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)

9. Extending SmartClient

Isomorphic 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.

The previous section (*Data Integration*) outlined the approaches to extending SmartClient on the *server*. This section 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:

Application Components	Controls	Editors	Forms
	Grids	Visualization	Containers
	Navigation	Collections	Design

| Application Services | Skinning | Layout | Localization |
| | Drag & Drop | Data Binding | Context Menus |

| Virtual Browser Layer | Rendering | Event Handling | Browser Utilities |
| | Communication | Timing | Accessibility |

| Foundation Layer | Class System | Object Utilities | Data Structures |
| | Language Extensions | Logging & Debugging | |

| Web Browser | **IE** Windows | **Netscape** Windows/Mac | **Mozilla/Firefox** Win/Mac/Linux/Solaris | **Safari** Mac OS |

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 sections 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. Here's 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 three example themes (aka "skins") for you to explore:

`standard`.......... similar to the Windows Classic theme

`Cupertino`........ similar to the Mac OS 9 theme

`SmartClient` ... a unique SmartClient 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"`.

       -or-

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

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:

- Locate the "SmartClient" skin under /isomorphic/skins and copy the contents of that entire directory into a new folder called "BrushedMetal".

- 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/")

- Delete the /isomorphic/skins/BrushedMetal/load_skin.js.gz and the /isomorphic/skins/BrushedMetal/skin_styles.css.gz files.

- 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.

- 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.

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:

standard button t

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:
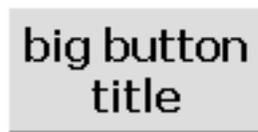
examples/custom_components/BigButton contains the code
for this example (including the "bigButton" CSS style definition).

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 <u>forums.smartclient.com</u>. 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 `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.

2. **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.

3. **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.

   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 typically implemented as custom "pickers" that the user can pop up from a picker icon next to a form or grid value.

To create a new form control:

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 `FormItem.iconClick` or `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 `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.

10. Tips

Beginner Tips

1. Pay extra attention to commas in your JS code.

Specifically in JS 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.

2. 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.

3. 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

1. 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.

2. 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.

3. 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.

4. Develop and deploy in browser compatibility mode, not "standards" mode.

SmartClient components automatically detect and adapt to the browser mode (as determined by DOCTYPE), providing consistent layout and rendering behaviors in both standards/strict and compatibility/quirks modes. However, the interpretation of "standards mode" varies across browsers, and changes across different versions of the same browser. *If you develop in "standards mode", the behavior of your application may change as users perform regular updates to their OS or browser.* "Standards mode" in most web browsers is *not*, as the name implies, a consistent standards-compliant mode.

Architecture Tips

1. **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.

2. **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.

3. **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.

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

End of Guide

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:

Web	smartclient.com
General	info@smartclient.com feedback@smartclient.com
Evaluation Support	forums.smartclient.com
Licensing	sales@smartclient.com

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