Sherlock Channels

(Legacy)

Internet & Web > Web Services

2007-04-09

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Accessing Channels 77

Introduction to Sherlock Channels

Important: Sherlock is unsupported in Mac OS X v10.5 and later.

The Sherlock application provides Macintosh users with a powerful tool for searching the Web. Users access different types of information in Sherlock through channels.

Prior to Mac OS X 10.2, channels in Sherlock were implemented as plug-ins that the user (or Sherlock) downloaded from the web and installed locally. These plug-ins provided a mapping for Sherlock to use in interpreting search results from an online source. The Sherlock application then merged the results from multiple sources and displayed them in a unified interface. Beginning with Mac OS X 10.2, Sherlock uses a powerful, new model for channels that gives channel developers more flexibility in how their data is displayed.

Organization of This Document

Sherlock channels provide a way to organize search results in a more intuitive and useful way. A channel implements a front-end interface for a Web-based search engine or other information database. However, unlike most browser-based searches, channels display the results using an Aqua interface and are capable of dynamically updating information.

Although it might seem like using Aqua to display search results would be a lot of work, Sherlock provides a significant amount of infrastructure to simplify the code required for your channel. Sherlock provides infrastructure for running the interface, dispatching events, managing network connections, parsing XML, and executing script code is transparent to the channel developer. With this infrastructure in place, channel developers are free to concentrate on the appearance and custom behavior of their channel.

This document describes how to create and manage a Sherlock channel and how to load the channel from a web page.

This programming topic contains the following articles:

For more information about Sherlock, and to obtain a copy of the Sherlock SDK, please visit the Sherlock Channel Development page: http://developer.apple.com/macosx/sherlock/.

Limitations

The channel architecture described in this document is available only versions of Sherlock that shipped in Mac OS X 10.2 or later. You cannot create channels for earlier versions of Sherlock using this architecture. If you want to build channels for earlier versions of Sherlock, you need to use the Sherlock plug-in architecture described in *Technical Note TN1141: Extending and Controlling Sherlock*.

The Sherlock channel architecture uses XML and supports the use of the JavaScript and XQuery languages for writing script code. Developing your channel interface requires Interface Builder with the Sherlock palette installed.

Architecture of Sherlock Channels

This article provides an overview of Sherlock channels and their architecture. It describes the components involved in creating channels, how they fit together, and how they work within the Sherlock environment.

Sherlock Overview

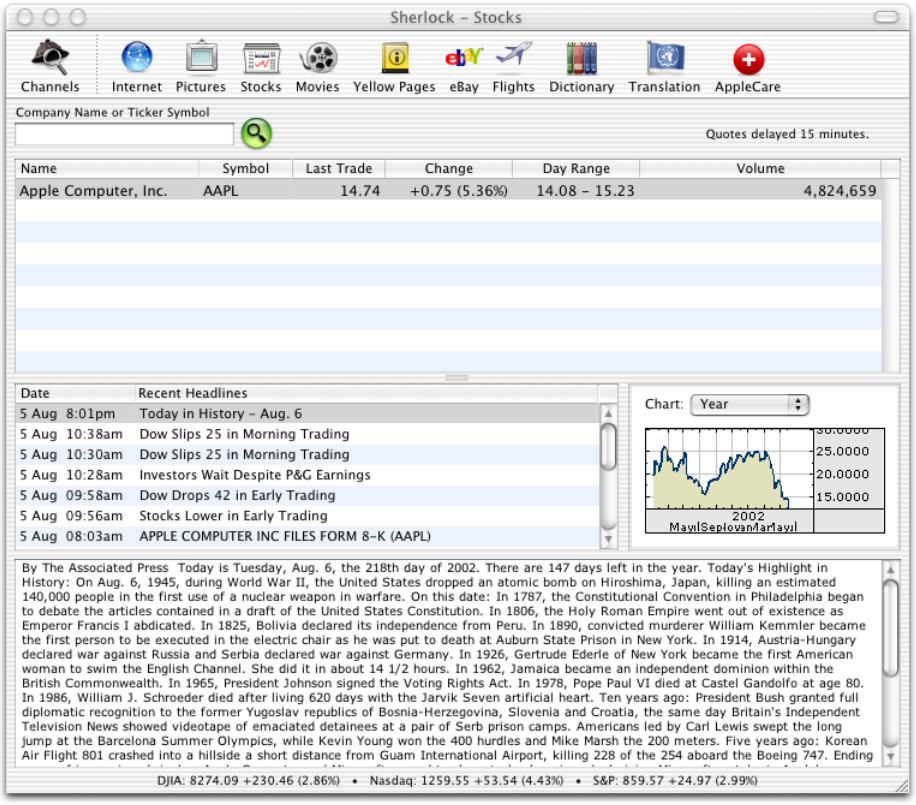
Sherlock is an application that incorporates Internet search capabilities into a flexible and extensible environment. The Sherlock application manages the infrastructure and support for **channels**, which do the work of gathering and displaying information.

A channel is a search-engine interface that uses the Sherlock infrastructure to access network-based resources and display the results to the user. Channels are not search engines in themselves; they take advantage of search engines on local intranets and the Internet to find information. However, channels can do more than just search for strings of characters. Channels can have a context in which to interpret the data they receive from a search engine. Using this context, the channel can then narrow the search criteria or perform additional searches to focus on information the user really wants.

You can develop channels to display a range of content, from movie showtimes to stock quotes, to yellow pages listings to news. You decide the level of detail you want to include in your channel and what context is required to achieve that detail. You then use that context to gather related information for the user. For example, a generic restaurant channel could use a postal code to provide restaurant listings in the user's area. A more complex restaurant channel could then use other web services to obtain driving directions or restaurant reviews. Your job as a channel developer is to set up the channel context to acquire the relevant data.

Figure 1 shows Sherlock's stock channel, which displays information about the user's selected stocks, current prices, and news. When the user selects a stock, the channel displays a list of headlines. Selecting a headline then displays the article associated with that headline. All of this information is gathered dynamically by the channel. The only information the user provides is the stock symbol or company name.

Figure 1 Sherlock stock channel



Most of window content you see in Figure 1 is provided by the channel. Sherlock manages the interface and code provided by the channel but is not responsible for providing the channel's behavior. The only portion of the window that Sherlock manages is the toolbar along the top edge. For more information on channel interfaces and how to create them, see "The Channel's Interface" (page 21).

Channel Architecture

The original architecture for Sherlock channels was relatively simple. The channel developer's main job was to provide a direct mapping between Sherlock data fields and the developer's search engine. The Sherlock application would then display search results as a weighted result set pulled from various sources. The new channel architecture gives you much more freedom to display information the way you want.

Overview

The goal of a Sherlock channel is to provide the user with relevant information for a specific topic. While you can use channels to act as a front-end for search engines, doing so does not take advantage of the power offered by Sherlock. The new architecture gives you a chance to create an intelligent search agent that gathers specific information and displays it in an intuitive way to the user.

The way you gather information in a channel is through a combination of search engines and web services. Search engines are a good starting point for finding basic text. However, web services are also becoming more prominent and are capable of offering more specific types of information. The script languages used by Sherlock make it easy to build XML queries and use them to communicate with SOAP services, among others.

Once you have the data you want, you must display it. Instead of the traditional search results table, Sherlock now supports the creation of custom user interfaces using Aqua controls. You can define an interface for your channel that is as complex or as simple as you want it to be. In either case, the result is a channel that behaves more like a Cocoa application than a web page of search results.

Channel Structure

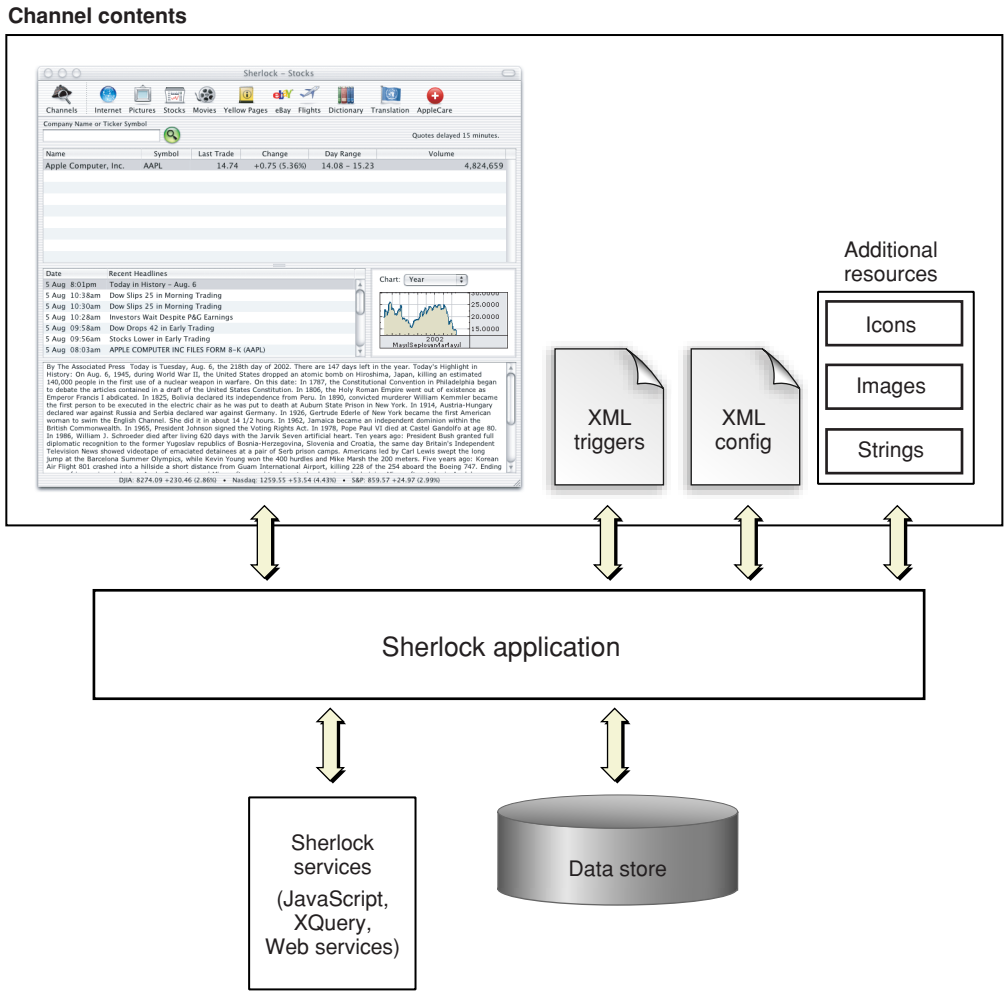
The way you create a channel has changed somewhat from previous versions of Sherlock. Whereas a channel used to consist of a mapping between the search engine syntax and the Sherlock syntax, channels now provide a user interface and script code to drive that interface. Sherlock still relies heavily on XML as a way of organizing the channel contents; however, channels also use the JavaScript and XQuery scripting languages to provide dynamic responses to data changes.

The channel itself now more closely resembles an application bundle containing resources and code files. The user interface for a channel is stored in a nib file that you create using Interface Builder. Your script code resides in an XML file, where it can be organized into small functions, called **triggers**, that respond to specific changes and events in your channel.

The Sherlock application provides the runtime environment in which channels operate. Sherlock provides a tremendous amount of infrastructure to support channels, including data storage, network services, and the script interpreters for your JavaScript and XQuery code. The most prominent piece of infrastructure is the **data store**, which acts as a repository for your channel's data. The data store also acts as the connection point between your channel's user interface and code, providing the place where the two exchange data.

Figure 2 shows the basic content of a channel and how that content relates to the Sherlock application and infrastructure. Each channel contains a nib file with the channel's user interface. Code resides in the XML triggers file. Sherlock uses the channel's XML configuration file to locate channel resources, including the nib file, XML triggers, and any additional resources. The Sherlock application coordinates interactions between your channel's files and the Sherlock infrastructure.

Figure 2 Basic channel structure



Understanding the Data Store

Understanding the data store and what it does is important for the development of Sherlock channels. The main function of the data store is, as its name implies, to store the data created by your channel or entered by the user. However, the data store has other behaviors that are important to the design and implementation of channels.

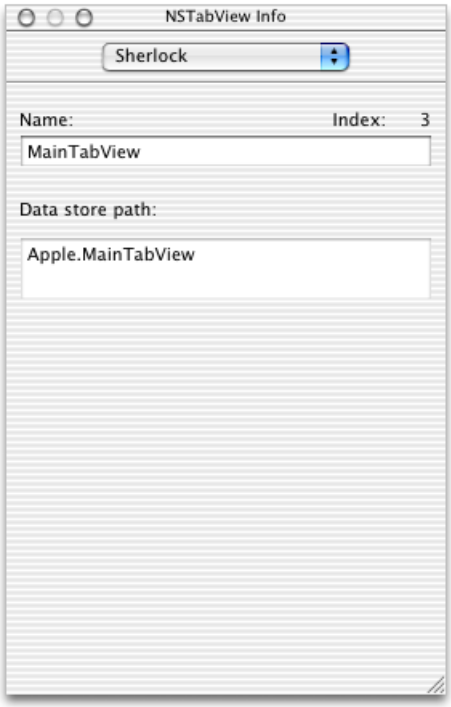
When a user selects your channel, Sherlock loads your channel's interface from the nib file you provide and incorporates it into the main Sherlock window. Because the Sherlock application runs the Sherlock window, your channel code has no direct connection to your user interface. Instead, Sherlock runs the interface and uses the data store as an intermediary for communications between the interface and your channel code.

Whenever the user interacts with your channel's user interface, Sherlock updates the data store to reflect the interaction. The data store is both a repository of information and a source of notifications for your channel. If the user types a value in a text field, Sherlock stores that value in an appropriate location in the data store and generates a notification that the data changed. If the user clicks a button, Sherlock does not modify any data store values, but it does generate a notification.

Information in the data store is organized on the concept of paths. A **path** is a string that uniquely identifies an object or property in the data store. Paths are not themselves objects that you manipulate. They are merely labels for data and services in the data store. You can use a path name to get or set the value for a control in your channel's interface. You can also send a notification to a particular path. The result of sending a notification is that Sherlock executes the script code for the trigger that associates itself with the notified path.

Every relevant view and control in your user interface must have a path. You assign path names using a special palette in Interface Builder. This palette (available as part of the Sherlock SDK) adds a new tab to the Info Window that lets you enter path name information for the controls and views of your interface. The path information is stored in your nib file and read by Sherlock when it loads your channel. Figure 3 shows the Info Window with the Sherlock tab.

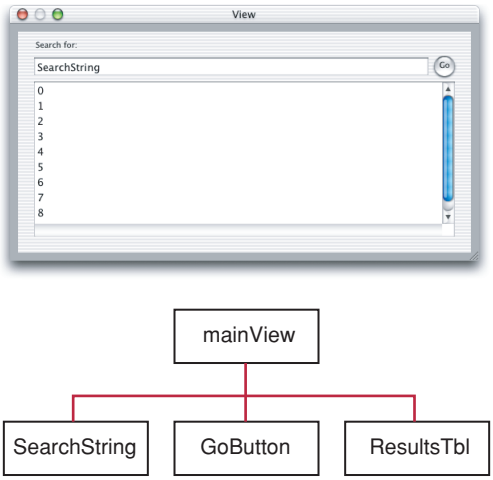
Figure 3 Sherlock tab of the Info Window



On the Sherlock tab, the control name is only part of the path name for that control. Your user interface must have a top-level view in which all other views and controls are embedded. The name of this view is prepended to the names of all other embedded views and controls. Sherlock includes only the top-level view in the path name for a control. It does not include the names of any other intervening views.

To access a property of a control, your script code must know the path to the control and the name of the desired property. Figure 4 shows a channel with a main view and several controls. The diagram to the right shows the organization and path names for each of the controls. To locate a property of a control, you would build a path with the name of the main view, the control name, and the property name, separating each name from the others with a period. For example, to access the data in the `SearchString` text field, you would create the path "`mainView.SearchString.objectValue`" and pass that path to the method for getting the data. For information on the properties defined for controls and views, see "Control Properties" (page 41).

Figure 4 Object containment hierarchy for a view



Sherlock defines specific paths for several predefined services and uses. Your channel can use these paths to store data persistently or to customize aspects of your channel such as its printing behavior. Some of these paths represent temporary variables, while others generate special notifications. You can define triggers for any of these paths to respond to its changes or notifications.

Channels do not share data with other channels or other instances of the same channel by default. Each window in Sherlock contains its own instantiation of a particular channel, and as such maintains its own separate copy of the data store. The only way to share data between channels is through the persistent storage paths of the data store. For more information, see "Persistent Storage Paths" (page 40).

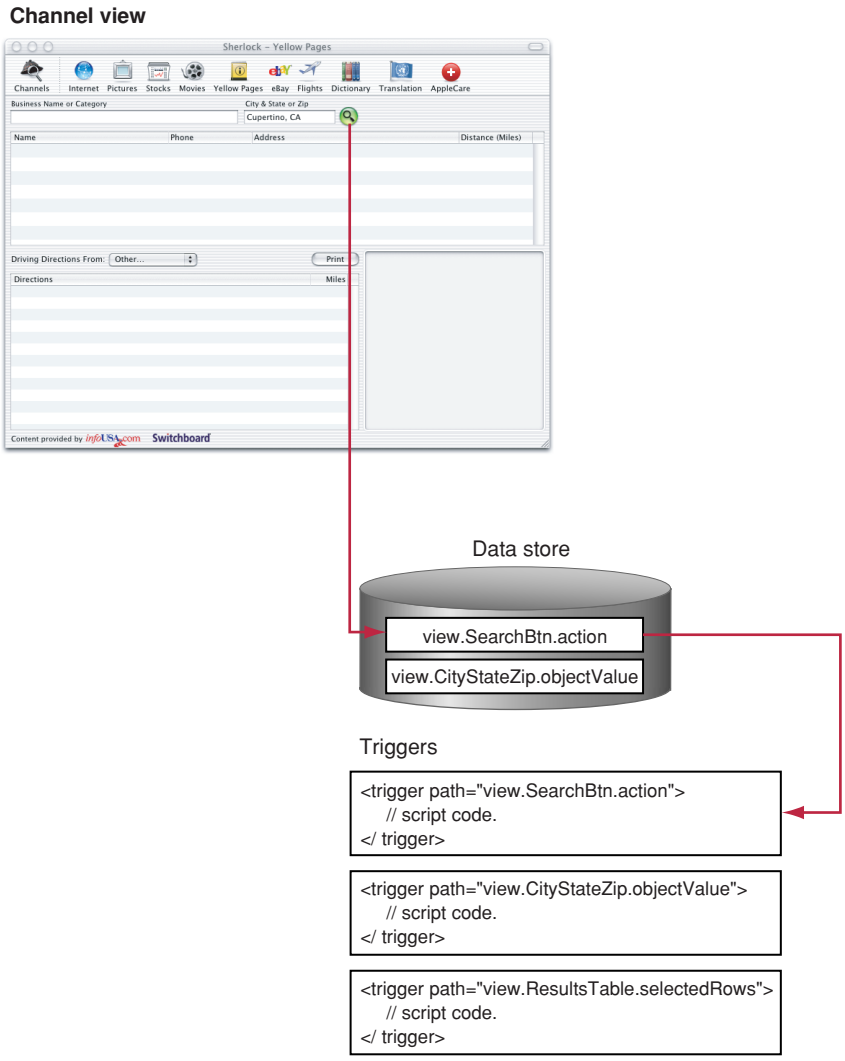
Understanding Triggers

A trigger is an event handler that responds to changes in the data store or to explicit notifications sent by Sherlock or your channel. You use triggers to respond to events in your channel's user interface and to handle other explicit or implicit notifications. The implementation of your trigger defines the behavior of your channel.

You define a trigger using the `<trigger>` tag in your channel's XML Triggers file. The content of this tag is the script code you want to execute. Every trigger must be associated with a specific path in the data store. Sherlock calls your trigger when a notification is sent to that path, whether because of a change in the data at that location or because of an explicit notification sent by Sherlock or your code. You specify the trigger's path using the `path` attribute. You can include additional attributes to specify other information needed by the trigger. See "Trigger Tag Syntax" (page 36) for more information.

Figure 5 shows the relationship between a channel's user interface and its triggers. The search button action has a specific path in the data store. When the user clicks the button, Sherlock generates a notification at that path to notify the channel that the button was pressed. Sherlock finds the trigger that responds to that path and executes its script code, passing in any additional information requested by the trigger.

Figure 5 Relationship between the channel interface and triggers



The script code for your triggers can be written using either JavaScript or XQuery, which is part of the XML Query specification. Each language has its strengths and weaknesses depending on the intended use. JavaScript is well suited for triggers that need to make dynamic decisions based on the current state of the channel. XQuery is well suited for performing operations that require complex text processing of data, such as generating query strings and parsing search results. Apple provides extensions to both languages for accessing Sherlock functionality. For more information on these extensions, see "Sherlock Reference" (page 33).

Deploying Channels

Unlike previous versions of Sherlock, which required the user to download a plug-in and install it locally, Sherlock now supports channel deployment over the Web. Web deployment simplifies the task of installing channels and also provides an easy way to update the channel content automatically. Sherlock caches the files of a channel and downloads them again when it detects any changes.

Sherlock also supports the grouping of channels together into a subscription. Subscriptions let the user enable or disable a group of channels all at once. They also make it easier for the user to download a group of channels initially. For example, Apple provides a subscription for three development-related channels. You can enable these channels from the Debug menu, when it is enabled.

Web Services

Although many developers will want to create channels for use in Sherlock, you can also create web services for use by channels. A web service is a routine or library of routines that performs a specific operation. Developing web services for your channels is a very simple process. All you do is place one or more routines in an XML file and deploy that file on your website.

Unlike many web services which run on the server, Sherlock web services run locally in the user's environment. Sherlock web services are essentially script files that the user's machine downloads and executes. A channel can include a web service file directly or it can use the `sherlock-function` routine from an XQuery script to access a specific routine of the web service.

In addition to developing your own web services, you can access existing web services by writing your own protocol wrappers. For examples of how to create and use web services, see the article "Using Web Services" (page 83).

Data Caching Strategies

In order to improve performance, Sherlock provides caching facilities to store channel files and data locally. Because caching may not be appropriate all the time, Sherlock provides channel developers with some flexibility over when to employ it. This section discusses the caching strategies available to developers of Sherlock channels.

Using Checkpoints

Although storing channel files on a server makes it easier to deploy updates to your channels, performance issues arise for users with slow network connections. Normally, when a channel is selected, Sherlock checks the modification dates of every file in the channel to see if any of the files changed. While it may not download every file, performing the network requests for these files can still take a noticeable amount of time. To eliminate this performance penalty, Sherlock supports the use of a checkpoint file to determine when a channel has changed.

If you implement checkpoint support, your channel configuration file acts as the checkpoint for your channel. The channel configuration file is the first file accessed by Sherlock when your channel is selected. This file contains a single `channel_info` tag with attributes telling Sherlock where to find your channel's resources. With checkpoints enabled, Sherlock compares the modification date of the cached file with the one on the server. If the modification dates are the same, Sherlock knows that the channel has not changed and uses the cached files whose download date is later than the checkpoint file.

If you implement checkpoint support in your channel, it is your responsibility to touch the modification date of your channel configuration file whenever you modify your channel. If you do not, users may not receive updates to your channel.

If your company provides a subscription for several channels, you can also use checkpoints in your subscription files to further reduce the number of network requests. Subscription files contain links to one or more channels. If the modification date of the cached subscription file differs from the server-based file, Sherlock then proceeds to check for changes to the channels of the subscription; otherwise, it assumes all channels are up-to-date.

For more information on the `channel_info` tag and enabling checkpoints in channels, see "Channel Information Tag Syntax" (page 33). For information on creating a subscription, see "Setting Up Subscriptions" (page 78).

Favoring Cached Data

Another place where Sherlock provides caching support is in the loading of files from the network. The `http-request` function in XQuery lets you request data from a network server. When you request a file for the first time using this method, Sherlock fetches it from the network and caches it. On subsequent requests, Sherlock compares the modification date of the cached file to the server file and returns the cached file if the dates are the same. However, you can eliminate this secondary network access by specifying some additional flags with the `http-request` function.

When you include the flag `FavorCache` in a request, Sherlock attempts to load the file from the cache. If the file is cached, Sherlock returns the cached copy without checking the network; otherwise, Sherlock loads the file from the network as usual. You can also use the `FavorCacheUpdate` flag to tell Sherlock to use the cached version now but to check the network for a newer version in the background. You may not get the newest file right away, but the next time you need it, the latest copy will be in the cache.

For more information on loading cached data files, see the description for "http-request" (page 64).

Version Information

As new features are added to the Sherlock development environment, developers may be wary of adding those features to a channel because of problems with backwards compatibility. Because not all users will have the same version of Sherlock installed on their system, channels need a way to identify code with new features and prevent them from running in unsupported environments. Sherlock handles this through the use of version attributes on a channel's tags. With version attributes, you can be sure that Sherlock runs new code in only those environments that support the new features.

Sherlock provides version information for three distinct portions of the system: the channel, the XQuery language, and the JavaScript language. Each tag that recognizes version information supports both a `version` and `maxVersion` attribute to specify the minimum and maximum supported versions, respectively. Table 1 shows the version information for Sherlock in different releases of Mac OS X.

Table 1 Sherlock version information

Item	10.2	10.2.2	10.2.3	10.2.5	10.3
Channel	1.0	1.1	1.2	1.3	1.4
XQuery	1.0	1.0	1.0	1.0	1.0
JavaScript	1.0	1.0	1.0	1.0	1.0

The channel version information is supported by the following tags:

- `<channel>` tags in a subscription file
- `<channel_info>` tag in a channel configuration file
- `<scripts>` tags in a code file

Although Sherlock performs version checks of tags internally, you can also get the channel version information programmatically if needed. To get the channel version, use the `channel-version` function from XQuery or the `ChannelVersion` function of the System object from JavaScript.

Important: The version of Sherlock that shipped with the original Mac OS X version 10.2 does not recognize the `version` attribute in `channel_info` tags. If you have a channel that requires Mac OS X 10.2.2 or later, you should include initialization code in your channel that checks the current channel version programmatically. If the channel version is "1.0", you can suggest upgrading to the latest system software using the System Update Preference panel.

The XQuery and JavaScript version information is checked by the following tags:

- `<initialize>` tag in a channel code file
- `<trigger>` tags in a channel code file
- `<script>` tags in a code file

To get the current JavaScript version, use the `Version` method of the System object. To get the current XQuery version, use the `version` function.

For more information on tag syntax and version attributes, see "XML Tag Syntax" (page 33).

Developing Channels

This article describes the contents of a Sherlock channel. Channels are implemented using a combination of XML code, script code, and various resources. A standard Sherlock channel consists of the following files:

- A nib file containing the channel user interface.
- A TIFF or icon file containing an image to display in the Sherlock toolbar.
- One or more XML files containing the script code for the channel's triggers
- An XML configuration file to tell Sherlock how to use the channel
- Localized versions of resources (including the Nib file, icons, and strings)

When you deploy your channel on a web server, you can organize these files in virtually any way you want. Your XML configuration file acts as a pointer to individual files, telling Sherlock where to find them. However, the section "Configuring Your Channel For Use" (page 27) provides some general guidelines on how to organize your files in an efficient way.

The sections that follow describe the contents of these files and how you use them to define your channel.

The Channel's Interface

The first step in creating a Sherlock channel is deciding the layout of your channel's interface. Unlike previous versions of Sherlock, you can now define a custom interface for your channel using the Interface Builder application and Aqua-style controls. With these tools, you can display search results in a more intuitive and user-friendly way. And because it is implemented using Cocoa, your channel's interface is much more responsive and flexible than before.

The main purpose of your channel's nib file is to provide the layout and path information for your interface. Unlike traditional Cocoa applications, Sherlock channels do not use much of the resource information provided by Interface Builder. For example, channels do not rely on outlets or actions. Instead, Sherlock identifies your views and controls using path names, which you assign in Interface Builder. As a result, you can use an empty project file for any new channels you create in Interface Builder.

The main interface for a channel is always a custom view you create in Interface Builder. Because it hosts several channels, the Sherlock application provides the main window for the channel content. This window displays the Sherlock toolbar and contains an area in which to display the currently selected channel. When a new channel is selected, Sherlock looks for a custom view in the channel's nib file, and when it finds it, loads that view into the main window.

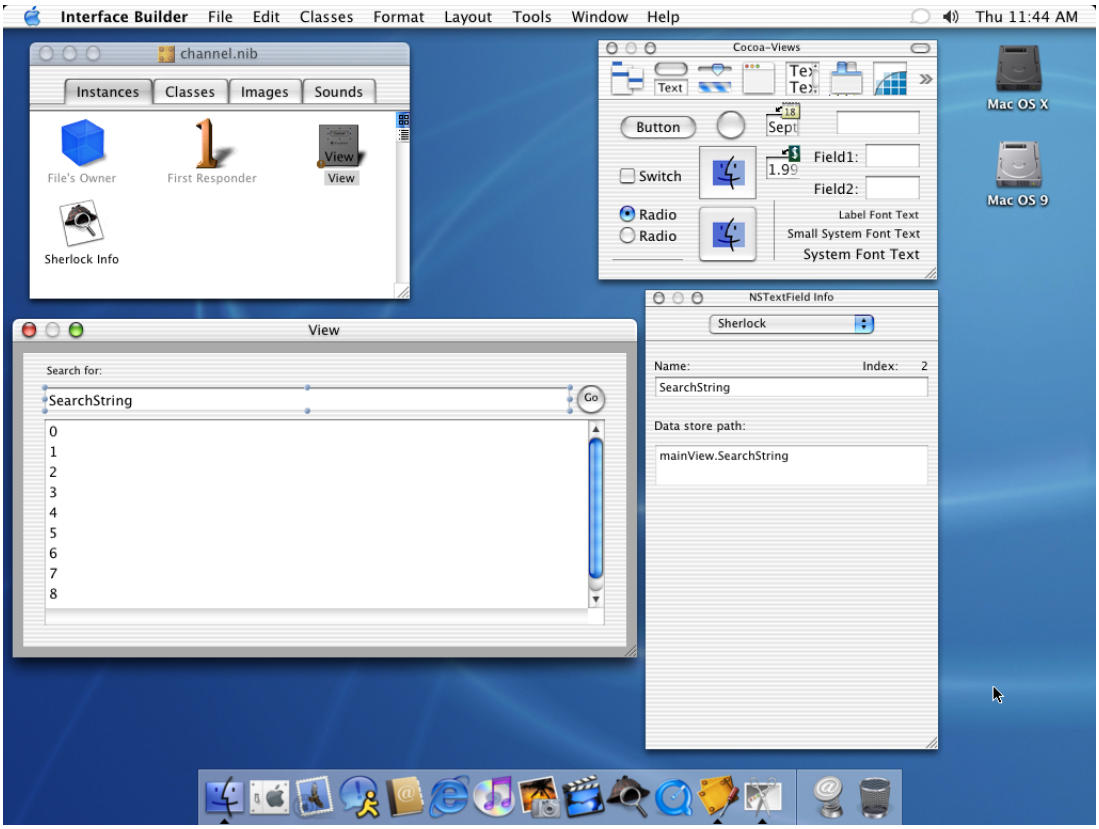
Your channel interface is not limited to a single view. You can define additional views for printing or for other versions of your channel. You can also define windows to implement sheets. However, if your channel's nib file contains multiple views, Sherlock needs to know which one to use in the main window. You tell Sherlock

which channel you want to use by assigning the name of the main view to the path `mainViewIdentifier` in your channel's initialization code. Sherlock checks this path and uses the value to load the view with the matching name. See "Initializing Your Channel" (page 24) for more information on channel initialization.

It is important to provide path names for each of the relevant views and controls in your channel interface. You assign path names to controls using the Sherlock tab of the Info Window in Interface Builder. You must provide path names for each top-level view and for any controls or subviews that you plan to use. The path of each top-level view defines the context for identifying the view and its contained controls in the data store. When referring to a control, the path name of the control always includes the name of the top-level view.

Figure 1 shows a sample Interface Builder project file with a custom channel view. The channel view contains a text field, button, and table used to gather search criteria and display the results. The Info Window on the right shows the path name information for the currently selected control. You need only enter the name of the individual control. Interface Builder constructs the full path name for you and displays it below the control name.

Figure 1 Creating your channel interface in Interface Builder



You do not need to add a path name for static text or image controls unless you plan to change the information displayed by that control from within your channel. Similarly, you do not need to assign path names to any views that do not contain additional controls. However, you should always remember to assign a name to the top-level view containing your channel interface.

Note: When assigning path names to controls, make sure you assign the name to the actual control and not to any parent view in which it's nested. For example, when naming an NSTableView, you must apply the path name to the NSTableView object, not its enclosing NSScrollView.

As you implement your channel interface, keep in mind how information flows in the interface. Channel script code reacts to changes in a control's data, so it is a good idea to spend time laying out the flow of information from one control to the next. Knowing this flow ahead of time makes it easier to implement your script code.

Once you are satisfied with your user interface, you can save your nib file and exit Interface Builder. The next stage in channel development is writing the XML and script code that defines the runtime behavior of your channel interface. This process is described in the section "Writing Your Channel Code" (page 23).

For an example of how to build a simple channel interface, see the article "Creating a New Channel" (page 73).

Writing Your Channel Code

Once you have defined the interface for your channel, you need to write the code to respond to events in that interface. Sherlock provides most of the infrastructure for managing your interface and informing your channel of changes. However, you are still responsible for implementing the specific behavior of your channel such as performing queries, organizing the resulting data, and updating the data store. You do this using triggers, which are described in the following sections.

About Triggers

Triggers make up the bulk of your channel's code. While you can use web services to perform specific tasks, triggers are how you receive notifications from Sherlock that something has happened. It is up to you to decide how you want to respond to each notification you receive.

The controls of your channel interface have properties that you can access from your triggers to get information about the control. You can gather information from these properties and use it to determine what actions to perform. You can write data to these properties to update your channel's interface. You can use these properties as targets for notifications and use them to execute other triggers.

Note: It is possible to supress the execution of triggers when you modify the data at a path. In JavaScript, you can call the `SetNoNotify` method of the `DataStore` object. In XQuery, you can add the `noNotify` attribute to your trigger tag. For information on the `DataStore` object, see "Sherlock Reference" (page 33).
.

Sherlock supports trigger scripts written using either JavaScript or XQuery. You can use one language or the other or you can mix calls between the languages using the objects and functions provided by Apple. For information on these languages, see "Sherlock Scripting Language Support" (page 29). For information on the JavaScript and XQuery language extensions, see "Sherlock Reference" (page 33).

For more information about triggers, see "Understanding Triggers" (page 16)

The XML Trigger File

Each channel has one main code file that contains the triggers for the channel. The content of this file is a set of structured XML tags organizing the triggers and initialization code. A basic channel definition file has the following structure:

Listing 1 Contents of a channel definition file

```
<?xml version="1.0" encoding="UTF-8"?>
<channel>
    <initialize language="JavaScript">
        // Initialize any data store variables here.
    </initialize>
    <triggers>
        <trigger language="JavaScript" path="URL.search">
            // Trigger script code
        </trigger>
        <trigger language="XQuery" path="Control.action">
            {-- Trigger script code --}
        </trigger>

        <!-- Additional trigger definitions -->
    </triggers>
</channel>
```

The `<channel>` tag is the top-level tag for your code file. All of your code must be nested inside this tag. The channel tag can contain an `<initialize>` tag with the channel's initialization code and a `<triggers>` tag with the channel's triggers.

The `<initialize>` tag provides a convenient way to initialize data store paths and configure your channel for first use. Sherlock executes the code in your initialization block the first time your channel is loaded in a new window. You can initialize data values, send notifications, read in values from persistent storage, or perform any other actions you need to prepare the channel for use. See "Initializing Your Channel" (page 24) for more information.

The `<triggers>` tag marks the beginning of the channel's trigger definitions. Nested inside this tag are one or more `<trigger>` tags defining the triggers for your channel. Each trigger has a path attribute specifying the data store path monitored by the trigger. Changes to the data at a path cause Sherlock to execute the trigger with the matching path attribute. Similarly, notifications sent by your script code to that path result in the execution of any matching triggers. See "Trigger Tag Syntax" (page 36) for more information.

Initializing Your Channel

When Sherlock first loads your channel, it executes a special block of code in your XML Trigger file. This block is identified by the `<initialize>` XML tag and contains script code to execute during the loading of your channel. You can use this initialization block to perform any initial setup required by your channel. For example, you can initialize the values of any controls or data variables. If your channel uses multiple views, you can also use this block to specify the channel's main view.

Sherlock executes your channel's initialization code the first time your channel is loaded into a window. Sherlock does not call your initialization code when the user toggles between channels in the same window. However, Sherlock does call your initialization code again if the user opens a new window.

The following example shows the initialization code for the Yellow Pages channel. The code enables printing for the channel and then proceeds to adjust the controls to their initial values. The code sets some display parameters, sets some button images, and tells the data store to send notifications whenever the main search attributes change. The code also sets the temporary variable `Defaults.cityStateZip` to its initial state. The value in this variable is used by the channel's triggers when the user does not specify a zip code, city, or state information.

Listing 2 Initializing a channel

```
<initialize language="JavaScript">
    DataStore.Set("mainViewIdentifier", "YellowPages");

    /* set up the data store to indicate that we want to handle printing */
    DataStore.Set("customPrint", 1);

    DataStore.Set("YellowPages.minViewSize", "{width=780; height=490}");
    DataStore.Set("Defaults.cityStateZip", "Cupertino, CA");
    DataStore.Set("YellowPages.CityStateZipField.updateValueOnTextChanged",
            true);
    DataStore.Set("YellowPages.MainQueryField.updateValueOnTextChanged",
            true);
    DataStore.Set("YellowPages.SearchButton.imageURL",
            "../shared/search.tiff");
    DataStore.Set("YellowPages.SearchButton.altImageURL",
            "../shared/searchDown.tiff");
    DataStore.Set("YellowPages.OpenLocations", false);

    DataStore.Set("YellowPages.PanX", 0);
    DataStore.Set("YellowPages.PanY", 0);
    DataStore.Set("YellowPages.MapZoom", 3);

    DataStore.Set("YellowPages.PanUp.imageURL", "panUp.tiff");
    DataStore.Set("YellowPages.PanDown.imageURL", "panDown.tiff");
    DataStore.Set("YellowPages.PanRight.imageURL", "panRight.tiff");
    DataStore.Set("YellowPages.PanLeft.imageURL", "panLeft.tiff");
    DataStore.Set("YellowPages.PanCenter.imageURL", "panCenter.tiff");

    /* Attribution */
    DataStore.Set("YellowPages.infousaImageButton.imageURL",
            "infousa.tiff");
    DataStore.Set("YellowPages.SwitchboardImageButton.imageURL",
            "Switchboard.tiff");
</initialize>
```

For more information on printing, see the article "Printing Your Channel's Content" (page 79). For information on the DataStore object, see "DataStore Object" (page 56).

Factoring Your Trigger Code

An important consideration when writing your trigger code is reuse. If you have a complex channel, you may want to be able to initiate a particular action in several different ways. For example, a user could trigger the action using a control in your channel interface while an HTML page could trigger the action using a Sherlock URL. It is a good idea to separate out reusable pieces of code into separate functions that you can call from multiple triggers.

When you factor code, you do not have to include all of your subroutines in your channel's main code file. Instead, you can declare your functions in a separate file and include it from your XML Trigger file. The content of an include file is usually only script code. To make it easy to include the file anywhere, you should always surround your code with a `<script>` tag containing a `language` attribute. The script tag ensures that the proper language is specified for your code. For example, if you create a file called `moreCode.xml` and use it to store some XQuery functions, the outline of your code file would look something like the following:

```
<scripts>
    <script language="XQuery">
        define function MyFunction($param1, $param2)
        {
            {-- Your code here. --}
        }
    </script>
</scripts>
```

To include this file in your XML Trigger file, you use the `<scripts>` tag in your `<triggers>` code block. You specify the file you want to load using the `src` attribute of the tag. The `src` attribute specifies the path to the include file, which can be a relative or fixed path. For example, to load the file `moreCode.xml` located in the subdirectory `services`, you would use the following code:

```
<triggers>
    <scripts src="services/moreCode.xml" />

    <!-- Additional triggers. -->
</triggers>
```

Localizing Resources

Sherlock supports channel localization using a mechanism similar to Mac OS X bundles. At the root level of your channel folder, you create subfolders for each of the localizations you support. The name of each subfolder consists of the two-letter ISO 639 language code followed by the `.lproj` extension. For example, the folder for English localized resources would be called `en.lproj`. Inside of each folder, you put the localized resources needed by your channel, including nib files, string resources, and image files.

When a user accesses your channel, Sherlock automatically checks the language preferences of that user and loads the appropriate nib file from your channel. To load localized resources from your script code, you can use the XQuery functions `localized-resource` and `localized-url`. To load localized resources from JavaScript, you must call these same functions using the XMLQuery object. See "XMLQuery Object" (page 57).

String resources reside in a property list file called `LocalizedResources.plist` inside of each localized resource directory. Sherlock requires that the name of your channel be included in this file as a localized string with the key `CHANNEL_NAME`. You can include additional strings, dictionaries, and arrays as needed by your channel. The following listing shows the strings defined by the Dictionary channel.

Listing 3 Defined strings from the Dictionary channel

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM
        "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
    <dict>
```

```
        <key>CHANNEL_NAME</key>
        <string>Dictionary</string>
        <key>SPELLING_SUGGESTIONS</key>
        <string>Spelling Suggestions</string>
        <key>AMERICAN_HERITAGE_DICTIONARY</key>
        <string>American Heritage Dictionary</string>
    </dict>
</plist>
```

To load a localized string, you use the `localized-resource` routine in XQuery. This routine accepts a key name and returns the string appropriate for the user's current locale settings. The following code load shows how to use this routine in your XQuery code.

```
let $columnTitle = localized-resource("SPELLING_SUGGESTIONS")
```

If you are writing your channel code using JavaScript, you can use the XMLQuery object to evaluate any XQuery expressions, as shown in the following example:

```
columnTitle = XMLQuery.localized_resource(""SPELLING_SUGGESTIONS");
```

For more information, see the description for "XMLQuery Object" (page 57).

Configuring Your Channel For Use

In order to make your channel available, you must tell Sherlock where to find it. Sherlock channels are distributed over the Web to make them easier to maintain and update. However, in order to distribute a channel over the web, Sherlock needs to know the location of your channel's code and resources. You do this by specifying a channel configuration file. Links that refer to your channel point to this file so that Sherlock can load the information it needs.

The channel configuration file is an XML file containing a single `<channel_info>` XML tag. The attributes of this tag identify the location of your channel's resource files relative to the configuration file itself. Listing 4 shows a sample channel directory structure with the channel's code and resource files. In this listing the `MyChannel.xml` file is the channel configuration file while `channel.xml` is the main code file containing the channel's triggers.

Listing 4 Channel directory structure

```
MyChannel/
    channel.xml
    en.lproj/
        channel.nib
        help/
            help.html
        localizedResources.plist
    mychannel.icns
    mychannel.tiff
MyChannel.xml
```

The following code listing shows the complete contents of the file `MyChannel.xml`. Sherlock loads this file and automatically interprets it as an XML tag. The attributes of the tag provide Sherlock with a way of uniquely identifying the channel, along with the location of the channel's resources, here shown as relative paths on the Web server. See "Channel Information Tag Syntax" (page 33) for a complete description of the `channel_info` tag and attributes.

Listing 5 Channel info tag

```
<!-- My channel info file -->
<channel_info
    identifier="com.company.MyChannel"
    icon_url="MyChannel/mychannel.tiff"
    channel_url="MyChannel/channel.xml"
    localized_base_url="MyChannel/"
    nib_url="channel.nib/objects.nib"
    icns_icon_url="MyChannel/mychannel.icns"
    help_file="help/help.html"
/>
```

In addition to configuring your channel, you may also want to provide a description of your channel for the channel management portion of the Sherlock window. To specify this string, add the `CHANNEL_DESCRIPTION` key to the `LocalizedResources.plist` file in each of your localized directories. The value of this key is the description of your channel. See "Localizing Resources" (page 26) for more information on localizing strings.

Once you have organized your project and created your configuration file, you can proceed to install your channel and debug it. For more information, see "Accessing Channels" (page 77).

Deploying Your Channel

The first step in using a channel is to make it accessible to users. Because Sherlock channels are small, they are ideally suited for deployment over the web. The minimal channel consists of two XML files, a nib file, a property list, and a TIFF image.

For testing purposes, you can deploy your web on your local machine using the personal web sharing feature of Mac OS X. With personal web sharing, you just need to drag your channel files into the current user's `Sites` folder to make the files accessible on the web. When you are satisfied that your channel works as expected, you can deploy it to a more public server and establish links to the channel's configuration file. For information on loading a channel from a URL, see "Loading a Channel From a URL" (page 77).

Regardless of where you deploy your channel, the channel configuration file must contain valid paths to the files of your channel. For more information on creating a channel configuration file, see "Configuring Your Channel For Use" (page 27).

Sherlock Scripting Language Support

Sherlock provides you with two different languages—JavaScript and XQuery—for writing your channel's script code. These languages provide you with both the power and flexibility to process data and manipulate your channel's contents dynamically. This article provides a brief introduction to each language and provides information about Apple's implementation.

Introduction to JavaScript

The JavaScript language was originally developed by Netscape as a way to implement dynamic HTML pages in a client's web browser. JavaScript is an interpreted language whose syntax resembles those of C in some aspects, but which also includes a mechanism for declaring and using objects. JavaScript is also known by the name ECMAScript and is typically used for writing short subroutines to manipulate browser data or respond to user interactions within web pages.

Sherlock supports JavaScript as one of the primary languages for writing triggers. Sherlock's uses a standard implementation of the JavaScript engine to interpret and execute script code in the Sherlock application environment. The JavaScript syntax is documented in numerous third-party books and is not covered in this article. However, developers familiar with the C and Java languages will find some familiar programming constructs.

Apple provides several predefined objects for JavaScript code to use. These objects are specific to the Sherlock channel environment and are described in more detail in "JavaScript Extensions" (page 54).

Introduction to XQuery

XQuery is a part of the XML Query specification currently being defined by the World Wide Web Consortium. The goal of XML Query is to provide a convenient way to extract data from real and virtual documents on the web. The medium for achieving this goal is XML and the script language used to manipulate the data is called XQuery.

A complete discussion of the XQuery language is beyond the scope of this article. For the latest information about XML Query, visit the World Wide Web Consortium website (`http://www.w3c.org/XML/Query`). This site includes information about the XQuery language and other current developments.

Apple provides several additions to the XQuery language that allow you to manage Sherlock resources. Apple's additions are documented in "XQuery Extensions" (page 58).

For new and experienced XQuery programmers, the following sections include some tips and techniques for using XQuery in your Sherlock channels.

Accessing the Data Store

Because XQuery triggers do not have access to the data store, you must use the `inputs` and `output` attributes of the trigger to get and set data store values. These parameters are described further in the section "Trigger Tag Syntax" (page 36).

An alternative to using the `output` attribute to return values is to return a dictionary of key/value pairs instead. When you return a dictionary of values from an XQuery trigger, the data store interprets the keys of the dictionary as paths. The data store then assigns the value for each key back to the corresponding path.

Commenting Out Text

Comments in XQuery begin with an open curly brace followed by two hyphens. To close the comment, use two hyphens followed by a close curly brace, as shown in the following example:

```
{-- Comments go here. --}
```

Support for Additional Data Types

Apple defines additional data types to supplement those provided by the XQuery language. Among the data types are the `data`, `dictionary`, and `url` types, which correspond to the Core Foundation types CFDataRef, CFDictionaryRef, and CFURLRef respectively.

The `dictionary` type is especially useful when writing XQuery-based triggers. To create a dictionary, you use the `dictionary` function provided by Apple and defined in "XQuery Extensions" (page 58). When you return a dictionary from an XQuery trigger, Sherlock interprets the dictionary keys as data store paths and proceeds to update the corresponding values with the dictionary data.

Accessing the Web

Apple provides several additions to the XQuery language that make it easy to access web-based resources. The additions are built on top of Apple's Web Foundation Framework and provide access to the web using the HTTP protocol.

To request a URL, you can use any of the following functions:

- `http-get`
- `http-head`
- `http-post`
- `http-request`

These functions take the requested URL and any parameters and use them to generate an HTTP request. They return a dictionary of keys identifying the returned data and status information.

Deciding Which Language To Use

When it comes time to write your trigger code, you need to choose which language to use. Most tasks can be performed using either JavaScript or XQuery; however, some tasks are better performed in one language than the other. Knowing which language to use for a given trigger can simplify the channel development process.

Common uses for the JavaScript language include executing conditional code, putting values in the data store, and opening URLs based on user actions.

Advantages of using the JavaScript language to develop triggers and services include:

- JavaScript triggers run synchronously, allowing for more dynamic processing of inputs.
- Data store variables can be retrieved and set directly using the DataStore object.
- JavaScript is an established language with numerous resources available for learning the language.
- JavaScript can use XQuery functions through the XMLQuery object.

Disadvantages include:

- Performance of JavaScript triggers is typically slower due to their synchronous execution.
- JavaScript does not have a strong set of tools for processing text and XML expressions.
- There is no direct way to initiate HTTP requests from JavaScript. They must be performed using the XQuery functions.

Common uses for the XQuery language include retrieving data from a URL and parsing XML/HTML to create a list of results.

Advantages of using the XQuery language to develop triggers and services include:

- The XQuery syntax is well suited for transforming data and processing XML expressions.
- XQuery triggers run asynchronously by default, providing better performance and the ability to process several expressions simultaneously.
- Apple-provided extensions make it easy to access web pages and Sherlock web services.

Disadvantages include:

- The XQuery specification is still in flux and subject to change.
- There are fewer resources available for learning the XQuery syntax.
- XQuery triggers cannot access the data store directly. Triggers must use the `inputs` and `output` attributes to move data back and forth

Supported XQuery Functions

Sherlock supports a subset of the XQuery functions defined at http://www.w3.org/TR/2002/WD-xquery-operators-20020816/ . The supported functions are listed here in functional groups.

Date, Time, and Duration

`current-dateTime`, `dateTime`, `date`, `time`, `duration`

Numbers

`number`, `ceiling`, `floor`, `round`, `sum`

Strings

`string`, `string-length`, `normalize-space`, `concat`, `contains`, `starts-with`, `ends-with`, `substring`, `substring-after`, `substring-before`, `translate`, `compare`, `lower-case`, `upper-case`, `replace`, `string-pad`, `match`, `matches`, `string-join`, `codepoints-to-string`, `string-to-codepoints`

Other

`document`, `count`, `local-name`, `boolean`, `false`, `true`, `not`, `empty`, `exists`, `distinct-values`, `distinct-nodes`, `insert`, `remove`, `sublist`, `min`, `max`, `to`, `item-at`, `index-of`, `deep-equal`

Deprecated XQuery Constructs in Mac OS X 10.3

In Mac OS X 10.3, Table 1 list identifies the XQuery constructs that are being deprecated. You should update your channel code to eliminate calls these functions.

Table 1 Deprecated XQuery constructs

Construct	Notes
`sortby` function	This function is being replaced by the “order by” expressions instead.
== and !== operators	Use the `is` and `isnot` operators instead.
XML constructs with non-quoted attributes	An example of this construct is `<a href={$link} />`. You must now include quotes around the attribute value, so that the above construct would now be `<a href="{$link}" />`.
Using uninitialized variables	If you use a variable that has not been defined or initialized with a value, you will get an error.
Comments	Using a close curly brace character `}` inside a comment now generates an error.

Sherlock Reference

This article provides a reference for the various extensions and predefined data paths provided by Sherlock.

XML Tag Syntax

Sherlock channels use XML tags to organize and identify channel resources. The following sections describe the syntax for the XML tags you can use with Sherlock.

Channel Information Tag Syntax

The channel configuration file constitutes the entry point to a channel. This file contains a single `channel_info` tag that identifies the key locations of the channel's resource files. Sherlock uses the attributes of this tag to identify the channel and to load its resources. This file can also act as a checkpoint for determining when the channel files have changed.

Table 1 describes the attributes of the `channel_info` tag.

Table 1 channel_info attributes

Attribute	Description
`channel_url`	Specifies the path to your channel's main code file. This is the XML file containing your channel's initialization code and triggers.
`check_point`	Specifies whether the file containing the `channel_info` tag should act as a check point for changes to the channel. If the value of this attribute is `true`, Sherlock compares the modification dates of the cached file and the file on the server. If they are the same, Sherlock assumes the channel has not changed and does not download any channel resources from the server. If the dates are different, Sherlock updates the channel resources from the server normally.
`help_file`	Specifies the location of a help file. The value of this attribute is the path to the HTML file relative to the localized resource directories in your channel folder. (Ignored on versions of Mac OS X earlier than version 10.3.)
`icns_icon_url`	Specifies the location of your channel's file-system icon (`.icns`) file. This icon is downloaded and used when the user attempts to make a shortcut to the channel in the file system. The icon file you specify must contain an icon that is 128 pixels by 128 pixels in size.

Attribute	Description
`icon_url`	Specifies the path to the icon (`.icns`) or image file (`.tiff`) you want to use for your channel's toolbar icon. Typically, this path is relative to the location of the channel configuration file. (A 32 pixel by 32 pixel TIFF image is preferred for this icon.)
`identifier`	Specifies a unique identifier for the channel. Sherlock uses this value to distinguish your channel from other channels. Typically, you encode your channel name and company domain name in this string using Java-style package naming conventions.
`localized_base_url`	Specifies the path to the parent directory containing your localized resource project directories. The directory you specify for this attribute should contain additional project directories for each language your channel supports. For example, if your project supports English and French locales, this directory would contain `en.lproj` and `fr.lproj` subdirectories.
`maxVersion`	Specifies the maximum channel version under which this trigger can execute. Use this attribute to prevent the loading and execution of a channel with a later version of Sherlock than the one supported. A value of "1.0" represents the channel version initially made available with Mac OS X version 10.2. (Note that support for this attribute was added in Mac OS X 10.2.2. Earlier versions of Sherlock will ignore the attribute.)
`nib_url`	Specifies the location of your channel's interface objects without any preceding path information. This string is comprised of the name of your nib file followed by `/objects.nib`. Sherlock uses this attribute together with the `localized_base_url` and current user's language settings to build a path to the appropriate nib file.
`version`	Specifies the minimum channel version required to run the channel. Use this attribute to prevent the loading and execution of a channel with an earlier version of Sherlock than the one supported. A value of "1.0" represents the channel version initially made available with Mac OS X version 10.2. (Note that support for this attribute was added in Mac OS X 10.2.2. Earlier versions of Sherlock will ignore the attribute.)

For more information on using the `channel_info` tag, see "Configuring Your Channel For Use" (page 27).

Channels Tags Syntax

The following listing shows the format of the tags in a channel subscription file. The `<channels>` tag encloses one or more `<channel>` tags, each of which contains a URL to a channel configuration file. The `<localized-strings>` tag is optional and used primarily for the localization of the channel name itself.

```
<channels name="subscription_name" [check_point=<"true" | "false">] >
    [<channel url="channel_url" [version="min_version"]
                maxVersion="max_version"] />]+
    [<localized-strings>
        <localized-string language="2-letter_iso_code"
                key="key_name"
```

```
                string="value_for_key" />
    </localized-strings>]
</channels >
```

The following example shows a modified version of the channel subscription file provided by Apple for Sherlock development channels. In this example, the file includes a localized string for the subscription name. Sherlock displays the localized name for the subscription in the application preferences window. This example also includes two channels that support specific versions of Sherlock.

```
<channels name="Developer">
    <channel url="channels/xquery.xml" version="1.0" maxVersion="1.0" />
    <channel url="channels/javascript.xml"/>
    <channel url="channels/htmlview.xml" version="1.1" maxVersion="1.1" />
    <localized-strings>
        <localized-string language="en" key="Developer"
                string="Apple Developer Channels" />
    </localized-strings>
</channels >
```

Script Tag Syntax

The `scripts` and `script` tags let you modularize your channel's source code and include additional web services. The `scripts` tag either loads a source file from server or encapsulates one or more `script` tags and makes the corresponding code available to your channel code. The `script` tag groups functions implemented using the same language into one location.

Table 2 describes the attributes of the `script` tag.

Table 2 script attributes

Attribute	Description
`language`	Specifies language used to implement the enclosed code. The value of this tag can be either `JavaScript` or `XQuery`.
`maxVersion`	Specifies the maximum XQuery or JavaScript version under which this script code can execute. Use this attribute to limit the execution of script code to specific versions of Sherlock.
`version`	Specifies the minimum XQuery or JavaScript version required to run the trigger. Use this attribute to limit the execution of a trigger to specific versions of Sherlock.

Table 3 describes the attributes of the `scripts` tag.

Table 3 scripts attributes

Attribute	Description
`maxVersion`	Specifies the maximum channel version under which the code in this script file can execute. Use this attribute to limit the execution of script code to specific versions of Sherlock. A value of "1.0" represents the channel version initially made available with Mac OS X version 10.2.

Attribute	Description
src	Specifies the location of the script file to include. The value of this attribute can be either a full or partial path to the target file.
version	Specifies the minimum channel version required to run the code in this script file. Use this attribute to limit the execution of script code to specific versions of Sherlock. A value of "1.0" represents the channel version initially made available with Mac OS X version 10.2.

The following example shows the use of the `scripts` and `script` tag to declare a function. This declaration could be placed inline with the channel's triggers or in a separate file.

```
<scripts>
    <script language="XQuery">
        define function HTMLToText($html)
        {
            $html/source(.)
        }
    </script>
</scripts>
```

If you were to place the preceding code in a separate file and include it, you would use another `scripts` tag to include it in the code file containing your triggers. The URL you specify can be either a full path to the server with the file, or a relative path if the file is on the same server.

```
<scripts src="http://www.mycompany.com/services/webservices.xml"/>
```

For more information on using the `scripts` and `script` tags, see "Factoring Your Trigger Code" (page 25).

Trigger Tag Syntax

Triggers are a combination of XML tags and script code that you include in your channel's XML Triggers file. A trigger is identified by the `<trigger>` XML tag. The attributes of this tag identify information about the trigger, such as the path it responds to, the language of the enclosed script code, and any other execution options. The content of the tag is the script code to execute in the language given by the tag attributes. Sherlock triggers support script code written in the JavaScript and XQuery languages. The basic syntax of the trigger tag is as follows:

```
<trigger language="trigger_language" path="data_store_path"
            [additional_trigger_attributes]>
    [script_code]
</trigger>
```

Important: The trigger tag and all attribute names are case-sensitive.

The following example shows a trigger definition from the Yellow Pages channel. This trigger receives the value from a text field and stores it in the channel's preferences file. The path attribute of the trigger tag identifies the data store path to which the trigger responds—a text field with the name `CityStateZipField` in this case. The inputs attribute is a way of initializing local script variables with values from the data store.

```
 <trigger language="JavaScript"
        path="YellowPages.CityStateZipField.objectValue"
```

```
        inputs="cityStateZip=YellowPages.CityStateZipField.objectValue">
    DataStore.Set("PERSISTENT.cityStateZip", cityStateZip);
    DataStore.Set("YellowPages.ClickAgaintext.stringValue", " ");
</trigger>
```

Common Trigger Attributes

Table 4 lists the `<trigger>` tag attributes you can use for all triggers. All attribute names are case-sensitive and optional unless otherwise noted.

Table 4 Common trigger attributes

Attribute	Description
`fullPathInput`	Binds this trigger's canonical full path to a local variable. For example, suppose your general script code assigns the value "`name`" to the path `Data.results[1].title`. If you define a trigger with the path `Data.results`, Sherlock binds the string "`Data.results.1.title`" to the local variable you specify for the `fullPathInput` attribute.
`fullPathValue`	Binds the value located at the trigger's path to a local variable. For example, suppose your general script code assigns the string "`name`" to the path `Data.results[1].title`. If you define a trigger with the path `Data.results`, Sherlock binds the string "`name`" to the local variable you specify for the `fullPathValue` attribute.
`inputs`	Specifies a binding between one or more data store paths and local script variables. Use this attribute to transfer values from the data store to your scripts. To specify more than one binding, separate the entries with a comma. This parameter is used primarily if you are writing your trigger code using XQuery, which does not have direct access to the data store. You can also use this attribute in JavaScript to retain the original value of a path whose contents may change.
`language`	Assign the language in which the trigger script code is written to this attribute. Currently, this value can be either `JavaScript` or `XQuery`. This parameter is required.
`maxVersion`	Specifies the maximum XQuery or JavaScript version under which this trigger can execute. Use this attribute to limit the execution of a trigger to specific versions of Sherlock.
`mutex`	Identifies a mutex for the trigger. Only one trigger at a time may run with a given mutex.
`name`	Specifies the name of the trigger. This attribute is used primarily for debugging purposes.
`path`	Assign a data store path to this attribute. The data store executes the trigger script when the value at the specified path changes or receives a notification. For a complete list of control attributes that you can use as paths, see "Control Properties" (page 41). This parameter is required.
`pathInput`	Binds the path of this trigger to the specified local variable. You can use the resulting variable in your script code to identify the data store path to which this trigger responds.

Attribute	Description
`subPathInput`	Binds this trigger's canonical subpath information to a local variable. The subpath information is the portion of a path that describes the instance and attribute information of the specific object being accessed. For example, suppose your general script code assigns the value "`name`" to the path `Data.results[1].title`. If you define a trigger with the path `Data.results`, Sherlock binds the string "`1.title`" to the local variable you specify for the `subPathInput` attribute.
`task`	Set this attribute to `true` to cancel any previously running instances of this trigger. This attribute is useful for a situation where you do not want to complete the current operation before moving to the next operation. For example, suppose the user is browsing a list of items in an NSBrowser control that updates its columns with child information. Without this attribute set, selecting an item would load all of the child information for that item before allowing a new selection. With this attribute set, selecting a new item would cancel the previous load operation and proceed to load the child information for the new item.
`timer`	Specifies that the trigger is to run at specified intervals. If this attribute is set to `true`, the data store uses the value at the trigger's path as the execution interval (measured in seconds). For example, if you associate a trigger with the path `Data.results` and the value at that path is 2, the data store executes the trigger every two seconds.
`version`	Specifies the minimum XQuery or JavaScript version required to run the trigger. Use this attribute to limit the execution of a trigger to specific versions of Sherlock.

JavaScript Trigger Attributes

Table 5 lists the `<trigger>` tag attributes available for triggers written using JavaScript. All attribute names are case-sensitive and optional unless otherwise noted.

Table 5 JavaScript trigger attributes

Attribute	Description
`async`	If present, executes the trigger's JavaScript code in parallel with the code of other triggers. This option is disabled by default, causing triggers to run synchronously.

XQuery Trigger Attributes

Table 6 lists the attributes available for triggers written using XQuery. All attribute names are case-sensitive and optional unless otherwise noted.

Table 6 XQuery trigger attributes

Attribute	Description
`append`	Contains a boolean value specifying whether the trigger should append the output results to the specified path, rather than replace the original contents.

Attribute	Description
noNotify	Prevents a notification from being sent when the output value is returned to the data store. Normally, return of an output value generates a notification because it involves changing a field value. The inclusion of this attribute suppresses that notification.
notify	Contains a comma separated list of data store paths to be notified upon completion of the trigger. You can use this attribute to trigger other actions.
output	Specifies a binding between the return value of the function and a data store path. When the function finishes executing, Sherlock sets the value of the specified data path to the return value of the function.
stream	Contains a boolean value indicating that you want the trigger to return a partial result set initially and write the rest of the data out asynchronously. This feature improves performance when a search may take time due to network or server delays.

The following example shows the use of several XQuery specific attributes. In this example, the trigger appends output data to the path "Stocks.SymbolTable.dataValue". To prevent delays, the data is streamed out and a notification is sent to the path "Stocks.action.updateQuotes" when all of the data has been returned.

```
<trigger path="Stocks.action.symbols" language="XQuery"
            inputs="symbols= Stocks.action.symbols,
            table=Stocks.SymbolTable.dataValue"
            output="Stocks.SymbolTable.dataValue"
            notify="Stocks.action.updateQuotes"
            stream="true" append="true">
let $newSymbols := for $symbol in $symbols where
            empty($table/symbol[. = $symbol])
    return dictionary(("symbol", $symbol), ("id", string(unique-id())))
return $newSymbols
</trigger>
```

Predefined Data Store Paths

The data store includes several predefined paths that you can use to access or store data. This section describes these paths and explains how to use them.

Nib File Installation

Sherlock defines several trigger paths for determining when a channel's interface is displayed or removed from the Sherlock window. Table 7 lists the paths along with a description of when the path is triggered.

Table 7 Nib file installation paths

Path	Description
.didInstall	Called immediately after the nib file contents are installed in Sherlock's window.

Path	Description
`.didRemove`	Called immediately after the nib file contents are removed from Sherlock's window.
`.willInstall`	Called immediately before the nib file contents are installed in Sherlock's window.
`.willRemove`	Called immediately before the nib file contents are removed from Sherlock's window.

Persistent Storage Paths

Sherlock uses the `PERSISTENT` path to manage data in the Sherlock preferences file for the current user. You can use this path to get or set channel-specific data you want to retain between Sherlock sessions. You can also use it to access data shared by multiple channels.

To store channel-specific data, specify the `PERSISTENT` base path followed by a unique path name to identify the stored data. For example, to save the postal code of the current user to the preferences file, you could use the following JavaScript code:

```
userPostalCode = 95014;
DataStore.Set("PERSISTENT.postalCode", userPostalCode);
```

Inside the preferences file, Sherlock creates a dictionary for each channel's individual preferences. Any data a channel writes to the `PERSISTENT` path are placed inside that channel's dictionary and are accessible only by that channel.

If you want to share persistent data with other channels, you must do so using the `PERSISTENT.SHARED` base path. This path defines an area of the preferences file where channels may share data they find useful. To store shared data, you must add your channel identifier and the path name for the data to the `PERSISTENT.SHARED` path name. For example, to store a value at the path `MyData` in the channel whose identifier is `com.apple.MyChannel`, you would use the following path:

```
PERSISTENT.SHARED.com.apple.MyChannel.MyData
```

All channels use the same path to access your channel's shared data. However, paths you create in the persistent storage space are accessible for reading and writing by your channel but are read-only to all other channels.

Printing Paths

Sherlock provides several specific paths to implement printing support for channels. The use of these paths is discussed in detail in the article "Printing Your Channel's Content" (page 79).

Table 8 Printing paths

Path	Description
`customPrint`	Set the value of this path to 1 if your channel supports printing using a custom view. You do not need to assign a value to this path if you want to use Sherlock's default printing support.

Path	Description
`mainViewIdentifier`	Use this path to distinguish your channel's main view. from any other views, such as a print view. The value at this path contains the name of the view.
`print`	Use this path to receive printing notifications.

URL Paths

Sherlock supports the automatic loading and execution of a channel using a browser URL. Sherlock defines a `URL` base path for accessing the query attributes of a URL. If a URL attribute has a value, Sherlock stores the value in the `URL` path under the name of the attribute. Attributes without values generate notifications instead.

For example, the following code shows a potential URL that can be sent to Sherlock:

```
sherlock://com.apple.yellowPages?query=sushi&zip=95120&search
```

When this URL is received, Sherlock writes each query attribute to the `URL` path. Query attributes are written in the same order as they appear in the query string. After all of the parameters are written, Sherlock sends a notification to the path `URL.complete` to signal the end of the parameter list. So, from the preceding example, Sherlock would perform the following equivalent operations (shown here as if they were part of a JavaScript trigger) to store the data:

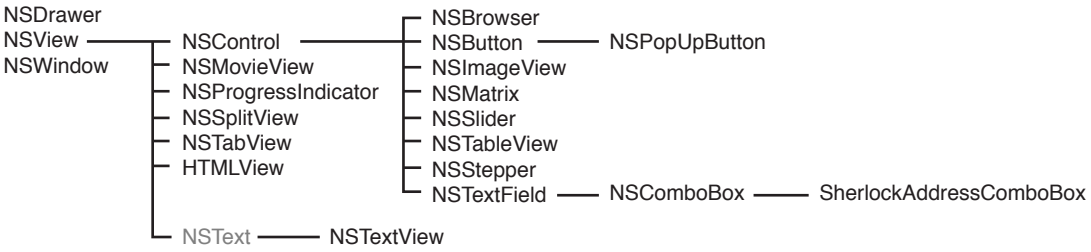
```
DataStore.Set("URL.query", "sushi");
DataStore.Set("URL.zip", "95120");
DataStore.Notify("URL.search");
DataStore.Notify("URL.complete");
```

For each data value it sets, Sherlock also generates a notification to any triggers monitoring that path. If your channel defined a trigger that monitored the `URL.query` path, that trigger would be called prior to any triggers for the `URL.zip` or `URL.search` paths.

Control Properties

The controls you use to define your channel interface have properties that you can access from your script code. Control properties are cumulative through the view and control hierarchies, that is, a specific instance of a control inherits the properties of its parent classes. Thus, an NSButton control inherits the attributes of NSControl and NSView. Figure 1 shows the inheritance hierarchy of the controls available to Sherlock channels.

Figure 1 Control inheritance hierarchy



The sections that follow describe the individual properties for each control.

HTMLView

HTMLView objects contain properties for rendering HTML content. HTMLView inherits properties from NSControl and NSView. Table 9 lists the path properties for HTMLView.

Table 9 HTMLView path properties

Property	Description
`baseURL`	Contains a string with the base URL to be prepended to any relative items or links in the HTML content.
`charset`	Contains a string with the name of the character set used by the HTML. Specify this value if the HTML does not already contain the information.
`followLinksInView`	Contains a boolean indicating whether new URLs may be loaded by clicking links in the view.
`htmlData`	Contains a string with the HTML data to render in the view. If you specify this value, do not specify a value for the `url` attribute.
`url`	Contains a string with the URL of the page to be rendered in the view. If you specify this value, do not specify a value for the `htmlData` attribute.

This control is supported only on Mac OS X version 10.3 and later.

NSBrowser

NSBrowser objects contain properties for accessing the column data of a browser (column view) control. NSBrowser inherits properties from NSControl and NSView. Table 9 lists the path properties for NSBrowser.

Table 10 NSBrowser path properties

Property	Description
`data`	Contains an array with one or more subarrays, each of which represents a column of the browser. Each subarray contains one or more dictionaries with key/value pairs associated with the current entry in the browser. Each dictionary can also contain the keys `VALUE` and `IS_LEAF`. The `VALUE` key contains the row-item text to display in the browser column. The `IS_LEAF` key identifies whether the current dictionary is a leaf.
`doubleAction`	Notified when the user double-clicks an item in the control. Your handler should check the selected cells to see what was clicked.
`selectedCells`	Contains an array with one or more arrays, each of which represents a column of the browser. The contents of each subarray are the selected cells in that column.

Although you can store hierarchical data using text and numbers in the data array of a browser, an easier way is to use dictionaries. When you specify a dictionary of items for a row, you can include a key called `IS_LEAF` to indicate that Sherlock should treat the dictionary as hierarchical data. When the key is set to true, Sherlock displays a right-arrow icon next to the row entry indicating there is more data.

NSButton

NSButton objects contain properties for getting and setting the display characteristics of the button. NSButton inherits properties from NSControl and NSView. You can use the `action` property inherited from NSControl to respond to button clicks. Table 11 lists the path properties for NSButton.

Table 11 NSButton path properties

Property	Description
`alternateImage`	Contains the button's alternate NSImage.
`altImageURL`	Contains a string or an array of strings with the URL's to use for the button's alternate image. If you specify an array of strings, Sherlock uses the first one if it is available; if not, it uses the second and the third and so on.
`bordered`	Contains a boolean indicating whether the button has a border.
`image`	Contains the button's NSImage.
`imageURL`	Contains a string or an array of strings with the URL's to use for the button. If you specify an array of strings, Sherlock uses the first one if it is available; if not, it uses the second and the third and so on.
`title`	Contains a string with the button's title.

NSComboBox

NSComboBox objects contain properties for getting and setting the items and text-completion characteristics of the combo box. NSComboBox inherits properties from NSTextField, NSControl, and NSView. Table 11 lists the path properties for NSComboBox.

Table 12 NSComboBox path properties

Property	Description
`comboBoxSelection-DidChange`	Path that is notified whenever the combo-box selection changes.
`completionFlags`	Contains an integer indicating the techniques used to automatically complete typed text entries. You can set this property to any combination of the following values: 1 - case-insensitive search 2 - literal search 3 - backwards search 4 - anchored search
`completes`	Contains a boolean value indicating whether the combo box automatically completes item names as the user types them.
`hasVerticalScrollBar`	Contains a boolean value indicating whether the combo box displays a vertical scroller.
`items`	Contains an array of strings representing the items in the combo box.
`selectAllOnFirst-MouseDown`	Contains a boolean value indicating whether the text in the combo box should be selected when the user clicks it. This value is false by default.
`selectedItem`	Contains a kCFNumberSInt32Type that represents the index of the selected item.
`showListOnBecome-FirstResponder`	Contains a boolean value indicating whether the combo box automatically shows its list when it becomes the first responder. This value is false by default.
`showListOnBeginEditing`	Contains a boolean indicating whether the combo box displays the list of choices as soon as a character is typed. This value is false by default.
`usesDataSource`	Contains a boolean indicating whether the combo box uses a data source to display its contents.

NSControl

NSControl defines common properties for the NSBrowser, NSButton, NSImageView, NSMatrix, NSSlider, NSTableView, NSStepper, and NSTextField controls. NSControl also inherits properties from NSView. Table 13 lists the path properties for NSControl.

Table 13 NSControl path properties

Property	Description
`action`	Path that is notified whenever the control's action occurs (such as when a button is clicked).
`actionURL`	Contains a URL to open when the control's action occurs. If this path contains an empty value, nothing happens.
`enabled`	Contains a boolean indicating whether the control is enabled.
`objectValue`	Contains a value of type `id` with the control's value. If you want to monitor the control's value, use this path instead of `stringValue`.
`stringValue`	Contains a string value with the control's value.

NSDrawer

NSDrawer objects contain properties to which you can send open and close notifications. Table 14 lists the path properties for NSDrawer.

Table 14 NSDrawer path properties

Property	Description
`close`	Post a notification to this path to close the drawer. This notification always occurs asynchronously.
`drawerDidClose`	Path that is notified when the drawer is closed. (Supported on Mac OS X version 10.3 and later.)
`drawerDidOpen`	Path that is notified when the drawer is opened. (Supported on Mac OS X version 10.3 and later.)
`open`	Post a notification to this path to open the drawer. This notification always occurs asynchronously.
`toggle`	Post a notification to this path to toggle the state of the drawer. This notification always occurs asynchronously.

NSImageView

NSImageView objects contain properties for accessing the image data and URL. NSImageView inherits properties from NSControl and NSView. Table 15 lists the path properties for NSImageView.

Table 15 NSImageView path properties

Property	Description
`cachedNSImage`	Contains the cached NSImage object. This value is read-only.

Property	Description
`data`	Contains the raw image data.
`shouldCacheNSImage`	Contains a boolean indicating whether the image should be cached in the data store.
`url`	Contains the URL that points to the image.

NSMatrix

NSMatrix objects contain the currently selected cells in the matrix. NSMatrix inherits properties from NSControl and NSView. Table 16 lists the path properties for NSMatrix.

Table 16 NSMatrix path properties

Property	Description
`cells`	Contains an array of cell objects.
`cells[n].imageURL`	Contains the URL for the image you want to display in the specified cell.
`cells[n].altImageURL`	Contains the URL for the alternate image you want to display in the specified cell.
`selectedCells`	Contains an array of zero-based indexes identifying the currently selected cells of the matrix.

NSMovieView

NSMovieView objects contain properties for loading and playing a movie. NSMovieView inherits properties from NSView. Table 17 lists the path properties for NSMovieView.

Table 17 NSMovieView path properties

Property	Description
`play`	Contains a boolean indicating whether or not the movie is playing. Setting this value to 1 initiates playback. Setting the value to 0 stops playback.
`showController`	Contains a boolean indicating whether or not to show the movie controls.
`URL`	Contains the URL for a network-based movie.

NSPopUpButton

NSPopUpButton objects contain properties identifying the items of the pop-up menu. NSPopUpButton inherits properties from NSButton, NSControl, and NSView. Table 18 lists the path properties for NSPopUpButton.

Table 18 NSPopUpButton path properties

Property	Description
`items`	Contains an array of strings, or an array of dictionaries with the keys `title` and `representedObject`. The `title` key identifies the string to be displayed. The `representedObject` key identifies the item represented by the corresponding row.
`selectedItem`	Contains the index of the selected item.

To insert a menu-item separator in the pop-up menu, add the string "SherlockMenuItemSeparator" to the array of strings in `items`. If you use an array of dictionaries to specify your menu items, put the "SherlockMenuItemSeparator" string in the `title` key and omit the `representedObject` key.

NSProgressIndicator

NSProgressIndicator objects contain properties for controlling the appearance of the progress bar. NSProgressIndicator inherits properties from NSView. Table 19 lists the path properties for NSProgressIndicator.

Table 19 NSProgressIndicator path properties

Property	Description
`bezeled`	Contains a boolean indicating whether the progress bar has a bezel.
`doubleValue`	Contains the current value of the progress indicator.
`indeterminate`	Contains a boolean indicating whether the progress bar is indeterminate.
`minValue`	Contains the minimum value of the progress indicator.
`maxValue`	Contains the maximum value of the progress indicator.
`startAnimation`	Send a notification to this path to start the progress bar animation.
`stopAnimation`	Send a notification to this path to stop the progress bar animation.
`usesThreadedAnimation`	Contains a boolean indicating whether the progress bar uses a threaded animation.

NSSlider

NSSlider has no accessible properties of its own, but it does inherit properties from NSControl and NSView. To set the minimum and maximum values of the slider, you must do so from Interface Builder.

NSSplitView

NSSplitView objects contain a property defining the minimum size of a split-view pane. NSSplitView inherits properties from NSView. Table 20 lists the path properties for NSSplitView.

Table 20 NSSplitView path properties

Property	Description
`minPositions`	Contains an array of numbers indicating the minimum number of pixels for each pane in the NSSplitView.

NSStepper

NSStepper has no accessible properties of its own, but it does inherit properties from NSControl and NSView. To set the minimum and maximum values of the stepper, you must do so from Interface Builder.

NSTableView

NSTableView objects contain properties identifying the appearance of the table's columns and rows, as well as the contents of the table. NSTableView inherits properties from NSControl and NSView. Table 21 lists the path properties for NSTableView.

Table 21 NSTableView path properties

Property	Description
`acceptableDropTypes`	Contains an array of strings listing the supported data types that can be dropped on the table view. The list of acceptable types includes `NSStringPboardType`, `NSTIFFPboardType`, and `NSURLPboardType`. (Supported on Mac OS X version 10.3 and later.)
`clickedColumnHeader`	Install a trigger on this path to receive notifications when a column header is clicked. This path contains the column identifier of the column that was clicked. If this path does not exist and the `disableSorting` attribute is not set to true, a default sorting routine is used for the table items.
`columns.<column_id>.cell.nibURL`	Contains a string identifying the location of the view objects to use for the column's cells. For example, if your view is defined in the nib file `MyView.nib`, you would specify the value "`MyView.nib/objects.nib`". The path you specify is relative to the XML Triggers file. If you specify a value for this property, you must also specify the value "`SherlockCell`" for the `columns.<column_id>.dataCellType` property.

Property	Description
`columns.<column_id>.dataCellType`	Contains a string identifying the type of information stored in the column. You can set this value to "`RankCell`" if you want the cell to contain ranking information for the row. Set this value to "`SherlockCell`" if you want to display a view object in the cell. If you use the "`SherlockCell`" value, you must specify the nib file containing your view in the `columns.<column_id>.cell.nibURL` property.
`columns.<column_id>.headerCell.objectValue`	Contains the title for the column with the specified column ID.
`columns.<column_id>.indicatorImage`	Contains a string or array of strings with the URLs to use for the title image of the column with the specified column ID.
`columns.<column_id>.noSort`	Contains a boolean indicating whether sorting is disabled for a specific column. (Supported on Mac OS X version 10.3 and later.)
`columns.<column_id>.sortKey`	Contains a string with the sort key for the column. If this attribute is not set, the column identifier is used as the sort key. (Supported on Mac OS X version 10.3 and later.)
`dataValue`	Contains an array of table values. If each item in the table is a string, then each array entry is a string. If the table has more than one column, each array entry contains a dictionary. The keys in each dictionary are the column identifiers that you set in Interface Builder.
`disableAutoSave`	Contains a boolean that indicates whether Sherlock is prevented from autosaving the column positions and widths. (Supported on Mac OS X version 10.3 and later.)
`disableSorting`	Contains a boolean that indicates whether sorting of the table's contents should be disabled. (Supported on Mac OS X version 10.3 and later.)
`doubleAction`	Install a trigger on this path to receive notifications when a row is double-clicked.
`doubleClickURL`	Contains a URL to open when the cell is double-clicked. If this path contains an empty value, nothing happens.
`droppedItems`	Contains an array of the items that were dropped onto the table view.
`firstVisibleRow`	Contains a zero-based integer indicating the first visible row in the table. The controls scrolls its contents as needed to make this row the first row.
`highlightedColumn`	Contains the column identifier of the currently highlighted column.
`pasteboardTypes`	Contains an array of dictionaries with the supported types of data for copy/drag operations. You must set the value for each supported pasteboard type to the data path containing the data. See the example that follows for more information.
`rowHeight`	Contains the height of each row as an integer.

Property	Description
`selectableColumns`	Contains an array of column identifiers indicating which columns can be selected.
`selectedRows`	Contains an array of zero-based integers indicating the currently selected rows.
`shouldDeleteSelection`	Contains a boolean indicating whether the user can delete rows of the table by selecting them and pressing the Delete key. If you do not explicitly assign a value of false to this property, NSTableView permits the user to delete rows from the table.
`trackFirstVisibleRow`	Contains a boolean indicating whether the value of the first visible row attribute should be updated when the table contents are scrolled.
`visibleDragColumns`	Contains an array of column identifiers indicating which columns can be dragged by the user. Note, that the column identifier is a separate attribute set in the Info Window of Interface Builder and does not necessarily equal the column title.

To specify pasteboard types, you associate an appropriate path name with the pasteboard type key. When the data is copied, Sherlock copies the data at the specified path to the pasteboard. For example, the following JavaScript code sets two different pasteboard types for a row. The first line sets the string version of the URL to the string object of the row. The second line sets the URL pasteboard data to the double-click URL for the row.

```
DataStore.Set(
        "Internet.SearchResultsTable.pasteboardTypes.NSStringPboardType",

"description.URL.objectValue");DataStore.Set("Internet.SearchResultsTable.pasteboardTypes.NSURLPboardType",
        "doubleClickURL");
```

Note that you should not specify the path to the row itself in the paths you specify. Sherlock assumes that you are copying from the current row automatically.

NSTabView

NSTabView objects contain a property identifying the currently selected tab. NSTabView inherits properties from NSView. Table 22 lists the path properties for NSTabView.

Table 22 NSTabView path properties

Property	Description
`selectedTabIdentifier`	Contains the identifier of the currently selected tab.

NSTextField

NSTextField objects contain properties for the control contents as well as for receiving notifications. NSTextField inherits properties from NSControl and NSView.

Table 23 lists the path properties for NSTextField.

Table 23 NSTextField path properties

Property	Description
`action`	Path that is notified whenever the field's user action occurs. The user action can be triggered on the end of editing or on pressing Return depending on how the control was set up in Interface Builder.
`htmlData`	Assign HTML text to the value at this path and it will be converted to a text representation and displayed in the text field.
`updateValueOn-TextChanged`	Contains a boolean indicating whether notifications should be sent as the user types. Notifications generated by this message monitor the `objectValue` path of the text field for changes.

NSTextView

NSTextView objects contain properties identifying their enclosed text. NSTextView inherits properties from NSView. Table 24 lists the path properties for NSTextView.

Table 24 NSTextView path properties

Property	Description
`drawsBackground`	Contains a boolean value indicating whether the control draws its own background.
`editable`	Contains a boolean value indicating whether the text is editable.
`htmlData`	Assign HTML text to the value at this path and it will be converted to a text representation and displayed in the text field
`objectValue`	Contains the value of the NSTextView.
`textChanged`	Path that is notified whenever the value of the NSTextView changes.
`valueAsString`	Contains the value of the NSTextView formatted as a string. The value at this path is read-only.

NSView

NSView objects define common properties for many controls and views. Table 25 lists the path properties for NSView.

Table 25 NSView path properties

Property	Description
`becomeFirstResponder`	Contains a boolean value indicating whether this view should be made the first to respond to events.
`hidden`	Contains a boolean value indicating the visibility of the view. You can set this value to change the state of the view. The default value of this attribute is `NO`.
`print`	Post a notification to this path if your channel supports custom printing and you received a print notification from Sherlock. This notification tells the view to print its contents.

NSWindow

NSWindow objects define properties for displaying and manipulating windows. Table 26 lists the path properties for NSWindow.

Table 26 NSWindow path properties

Property	Description
`beginSheet`	Post a notification to this property if you want to display the specified window as a sheet of the current channel window.
`close`	Post a notification to this path to close the specified window.
`display`	Post a notification to this path if you want to force the window to redraw itself entirely.
`endSheet`	Post a notification to this path to dismiss a sheet posted with the `beginSheet` notification.
`flushWindow`	Post a notification to this path if you want to flush the window buffer to the screen.
`hasShadow`	Contains a boolean value that indicates whether the window has a shadow.
`makeKeyAndOrderFront`	Post a notification to this path to make the window the key window. This notification also moves the window in front of any other windows.
`title`	Contains a string with the window title.

You typically use NSWindow objects defined in your nib files as sheets for your main channel window. To display a sheet, you send a `beginSheet` notification to the target window. For example, to display a sheet named `MySheet` when the user clicked a button, you would use the following code:

```
<trigger path="MyChannel.DisplaySheetBtn.action" language="JavaScript">
    DataStore.Notify("MySheet.beginSheet");
</trigger>
```

To respond to events in the sheet, you would define additional triggers to handle the sheet's controls, just as you do for your channel controls. For example, to respond to an OK button on the sheet, you would use code similar to the following:

```
<trigger path="MySheet.OKButton.action" language="JavaScript">
    DataStore.Notify("MySheet.endSheet");
</trigger>
```

SherlockAddressComboBox

SherlockAddressComboBox objects contain properties for displaying address information from the user's Address Book database. This control displays an NSComboBox control that can be used to enter or select address information. The control keeps a list of the 10 most recently used addresses for auto-completion purposes. SherlockAddressComboBox inherits properties from NSControl and NSView. Table 27 lists the path properties for SherlockAddressComboBox.

Table 27 SherlockAddressComboBox path properties

Property	Description
`autoCompletionTypes`	Contains an array of one or more of the following strings: `address`, `addressTitle`, `cityState`, `cityStateZip`, or `custom`, Each string represents a type of information to auto-complete for the user. Types are auto-completed in the order they are listed in the array. If you specify the `custom` string, you must provide a set of custom items to auto-complete in the `customItems` attribute.
`autoPopupToShow-Matches`	Contains a boolean indicating whether the combo box should automatically display matches.
`customItems`	Contains an array of dictionaries. Each dictionary contains information about the items to display in the popup and must contain the following keys: `title` and `representedObject`. The `title` key contains the string to be used for auto-completion. The `representedObject` key contains the object representing the particular item.
`maxAutoComplete-Matches`	Contains a number indicating the maximum number of auto-completed matches.
`representedObject`	Contains the object associated with a custom popup item.
`resultAddress`	Contains a dictionary with the auto-completion results. This path will always contain a dictionary with one or more of the following keys set to a non-nil value: `Street`, `City`, `State`, `ZIP`, `Country`, `CountryCode`, `UserText`, `representedObject`, The values for all of these keys are strings, except for `representedObject`, which is a custom popup item.
`UserText`	Contains the objectValue of the NSComboBox in situations where the controller cannot determine that the text is either an address in the user's address book or a city/state/zip combination.

This control is supported only on Mac OS X version 10.3 and later.

JavaScript Extensions

Sherlock comes with an implementation of the JavaScript language that you can use to write your triggers. In addition to the standard features available with JavaScript, Sherlock defines several additional objects that you can use in your trigger code. These objects are declared implicitly by Sherlock for you, so you do not need to create or declare these objects directly in your scripts.

AddressBook Object

The AddressBook object provides access to the local Address Book database. You can use this object to add new addresses to the database and search for existing addresses. Table 28 lists the methods defined by the `AddressBook` object.

Table 28 AddressBook object methods

Method Syntax	Descriptions
`Object Add(address)`	Creates a new record in the Address Book database. The `address` parameter is a dictionary containing the address information you want to add. The method returns a string containing the unique ID for the new record. See "Address keys" (page 54) for a list of valid keys.
`Object AddressWithUID(uid)`	Returns a dictionary containing address information for the record with the ID specified in the `uid` parameter. See "Address keys" (page 54) for a list of keys available in the returned object.
`Object Search(params, options)`	Searches the Address Book database for a specific address. The `params` parameter contains a dictionary with the address search strings. (See Table 29 for the address keys.) The `options` parameter contains a dictionary with the options to apply to the address search strings. See "Address keys" (page 54) for a list of valid keys.

Address keys

Table 29 lists the keys used to identify address information in the methods of the `AddressBook` object.

Table 29 Address keys

Key	Value
`Address`	Contains a dictionary with one or more of the keys `Home` and `Work`. Each of these keys, in turn, contains a dictionary with the following keys: `Street`, `City`, `State`, `ZIP`, `Country`, and `CountryCode`. All of these keys contain strings. The value for the `CountryCode` key is a two-letter country code.
`Email`	Contains a dictionary with one or more of the keys `Home` and `Work`. Each of these keys is a string containing the appropriate email address.

Key	Value
Phone	Contains a dictionary with one or more of the keys Home, Work, Mobile, Main, HomeFAX, WorkFAX, or Pager. Each of these keys contains a string with the appropriate phone number.
First	Contains a string with the first name of the individual.
Last	Contains a string with the last name of the individual
FirstPhonetic	Contains a string with the phonetic first name of the individual.
LastPhonetic	Contains a string with the phonetic last name of the individual.
Organization	Contains a string with the company name associated with the address.
JobTitle	Contains a string with the job title of the individual.
HomePage	Contains a string with the URL of the individual's home page.
Birthday	Contains a date with the birthday of the individual.
Note	Contains a string with any notes associated with the address.
MiddlePhonetic	Contains a string with the phonetic middle name of the individual.
Title	Contains a string with the title of the individual.
Suffix	Contains a string with any suffixes associated with the individual's name.
Nickname	Contains a string with the nickname of the individual.
MaidenName	Contains a string with the maiden name of the individual.
UID	Contains a read-only string with the unique ID of the Address Book record.
Modification	Contains a read-only date of the last time the record was modified.
Creation	Contains a read-only date of when the record was created.

Search Options

The options parameter of the Search method accepts a dictionary with two keys: conjunction and comparison. The value of the conjunction key indicates how to combine search parameters. You can specify either the string "and" or "or" for this value to indicate all parameters must be met or any of the parameters must be met, respectively.

The comparison parameter indicates the way in which search parameters are compared against database entries. All search parameters are compared using the same comparison operator. Table 30 lists the valid values for the comparison key.

Table 30 Comparison key values

Value	Description
`Equal`	Perform an equality check using a case-sensitive comparison.
`NotEqual`	Perform an inequality check using a case-sensitive comparison.
`LessThan`	Look for values less than the specified value.
`LessThanOrEqual`	Look for values less than or equal to the specified value.
`GreaterThan`	Look for values greater than the specified value.
`GreaterThanOrEqual`	Look for values greater than or equal to the specified value.
`EqualCaseInsensitive`	Perform an equality check using a case-insensitive comparison. (Strings only)
`ContainsSubString`	Look for values that contain the specified substring. (Strings only)
`ContainsSubString-CaseInsensitive`	Look for values that contain the specified substring using a case-insensitive comparison. (Strings only)
`PrefixMatch`	Look for values that start with the specified string. (Strings only)
`PrefixMatchCaseInsensitive`	Look for values that start with the specified string using a case-insensitive comparison. (Strings only)

DataStore Object

The DataStore object provides access to the channel's field data and coordinates the transmission and receipt of notifications. You use this object in your script code to get or set data store values. You can also use it to generate explicit notifications from your script code. Table 31 lists the methods defined by the `DataStore` object.

Table 31 DataStore object methods

Method Syntax	Description
`void Append(path, value)`	Appends a value to the specified path in the data store. When appending data to a path, Sherlock treats the contents of the path as an array, with each appended piece of data a separate array entry.
`Object Get(path)`	Gets a value from the specified path in the data store.
`void Set(path,value)`	Sets the value at the specified path in the data store. This method generates a notification event for the specified path.
`void Notify(path)`	Sends a notification event to the specified path in the data store.
`void SetNoNotify(path, value)`	Sets the value at the specified path in the data store, but does not generate a notification event for the path.

For example, to get the value at the path `MyPath.MyTextField.objectValue`, you would use the following JavaScript code:

```
value = DataStore.Get("MyPath.MyTextField.objectValue");
```

System Object

The System object provides some useful utilities, which are listed in Table 32.

Table 32 System object methods

Syntax	Description
`Object AllCountries ()`	Returns an array of dictionaries containing country information from the system. Each dictionary contains an `isoCode` key and a `name` key. The value for the `isoCode` key is the two-digit ISO country code. The value for the `name` key is the country name.
`type ChannelVersion()`	Returns the current channel implementation version. Sherlock uses the channel version to determine whether to load or execute XML tags that have `version` and `maxVersion` attributes.
`Object CountriesForCurrentLanguage ()`	Returns an array of dictionaries containing the country information for countries whose language matches the current user's language settings. Each dictionary contains an `isoCode` key and a `name` key. The value for the `isoCode` key is the two-digit ISO country code. The value for the `name` key is the country name.
`void OpenURL(url)`	Tells the system to open a URL with the appropriate application, as defined by the system defaults. This method supports the `addressbook`, `help`, `http`, `https`, `ical`, `mailto`, `sherlock`, and `webcal` schemes.
`Object UserPreferredLanguages ()`	Returns an array of strings representing the user's current language preferences. Entries in the array are sequential based on the order specified in the user's system preferences.
`type Version ()`	Returns the version of Sherlock's JavaScript implementation.

XMLQuery Object

The XMLQuery object provides an interface for calling XQuery functions from JavaScript. This object provides access to all of the built-in XQuery functions as well as to the Apple-specific additions. Each XQuery function is implemented as a method of the XMLQuery object. For example, to call the `url` function, you would use the following syntax:

```
urlObject = XMLQuery.url("http://www.apple.com");
```

Due to JavaScript naming conventions, when you call an XQuery function whose name contains a hyphen (-) character, you must replace the character with an underscore (_) instead. For example, to call the `convert-entities` function from JavaScript, you would use the following syntax:

```
outString = XMLQuery.convert_entities("&amp;");
```

XQuery Extensions

Sherlock comes with an implementation of the XQuery language that you can use to write your triggers. In addition to the XQuery functions, Sherlock defines several additional functions that you can use in your trigger code. The following sections describe the purpose and syntax of each function.

base-url

The `base-url` function returns a URL to the directory containing the file that called this method. The syntax for this function is as follows:

```
url base-url()
```

This function returns the location of the code file that called this method.

base64-decode

The `base64-decode` function decodes a base-64 encoded block of data into a string. The syntax for this function is as follows:

```
data base64-decode(source)
```

Base-64 encoding packs three 8-bit bytes into four 7-bit ASCII characters. If the number of bytes in the original data is not divisible by three, "=" characters are used to pad the encoded data. This function reverses the process to yield the original string.

base64-encode

The `base64-encode` function encodes the source data using base-64 encoding. The syntax for this function is as follows:

```
data base64-encode(source)
```

Base-64 encoding packs three 8-bit bytes into four 7-bit ASCII characters. If the number of bytes in the original data is not divisible by three, "=" characters are used to pad the encoded data.

For example, the following function call would yield an encoded value of "`MQ==`":

```
base64-encode("1")/string()
```

channel-version

The `channel-version` function returns a CFNumberRef with the current channel implementation version. The syntax for this function is as follows:

```
number channel-version()
```

charset-encoding

The `charset-encoding` function returns the encoding index for the specified IANA character set name. The syntax for this function is as follows:

```
number charset-encoding(encoding)
```

The *`encoding`* parameter contains the character set name.

charset-name

The `charset-name` function returns the IANA character set name corresponding to the specified index. The syntax for this function is as follows:

```
string charset-name(encoding)
```

The *`encoding`* parameter is an integer specifying the string encoding.

convert-entities

The `convert-entities` function converts HTML entity characters in a string to their equivalent display characters. The syntax of this function is as follows:

```
string convert-entities(source)
```

The *`source`* parameter contains a text string. Any characters of the form `&<code>;` where `<code>` is an entity name are converted to display characters. For example, the string `&` is converted to the ampersand character "&". This function returns a new string with the converted characters.

convert-html

The `convert-html` function converts HTML entity characters to displayable text and normalizes the spaces in the string. The syntax of this function is as follows:

```
string convert-html(source)
```

The *`source`* parameter contains a string with HTML text. This function converts entity characters to their displayable forms and normalizes whitespace characters. This function returns a new string with the displayable text.

In the following example, the call to `convert-html` returns the string "`&<a href=x.gif>picture</a>`".

```
convert-html(" &amp;<a href=x.gif>picture</a>     ")
```

curl

The `curl` function performs an HTTP GET request using `curl` to initiate the request. The syntax for this function is as follows:

```
data curl(source)
```

The *`source`* parameter contains the URL to request.

data

The `data` function builds and returns an object of type CFDataRef. The syntax of this function is as follows:

```
data data(theData)
```

data-length

The data function returns the number of bytes in a data block. The syntax of this function is as follows:

```
number data-length(theObject)
```

data-match

The `data-match` function searches a data block for the specified starting and ending patterns and returns the first match it finds. The syntax for this function is as follows;

```
data data-match(source, matchStart, matchEnd, includeStart, includeEnd)
```

The *`source`* parameter contains the data block you want to search. The *`matchStart`* parameter contains the initial match data. Once found, the match continues until the data in the *`matchEnd`* parameter is found. This function returns a data object with the matched contents.

The *`includeStart`* and *`includeEnd`* parameters tell this function whether to include the starting and ending match data in the returned string. These parameters are set to false by default and may be omitted.

The following example returns the data "0x3334", which represents the last two bytes of the data block. Only the last two bytes are returned because false is specified for the *`includeStart`* parameter.

```
data-match("1234","2","4",false(),true())
```

data-match-all

The `data-match-all` function searches a data block for the specified starting and ending patterns and returns a list of matches. The syntax for this function is as follows:

```
seq data-match-all(source, matchStart, matchEnd, includeStart,
        includeEnd)
```

The *source* parameter contains the data block you want to search. The *matchStart* parameter contains the initial match data. Once found, the function accumulates data until the *matchEnd* parameter is found. This function returns a data object with the matched contents.

The *includeStart* and *includeEnd* parameters tell this function whether to include the starting and ending match data in the returned string. These parameters are set to false by default and may be omitted.

The following example returns the data "0x3334, 0x3534", which matches the two patterns within the data block. Only the last two bytes of each match are returned because false is specified for the *includeStart* parameter.

```
data-match("1234 1254","2","4",false(),true())
```

data-match-ignore-case

The `data-match` function searches a data block for the specified starting and ending patterns and returns the first match it finds. The syntax for this function is as follows:

```
data data-match-ignore-case(source, matchStart, matchEnd, includeStart,
        includeEnd)
```

The *source* parameter contains the data block you want to search. The *matchStart* parameter contains the initial match data. Once found, the function accumulates data until the *matchEnd* parameter is found. Matches do not take case into account. This function returns a data object with the matched contents.

The *includeStart* and *includeEnd* parameters tell this function whether to include the starting and ending match data in the returned string. These parameters are set to false by default and may be omitted.

data-match-ignore-case-all

The `data-match-ignore-case-all` function searches a data block for the specified starting and ending patterns and returns a list of matches. The syntax for this function is as follows:

```
seq data-match-ignore-case-all(source, matchStart, matchEnd,
        includeStart, includeEnd)
```

The *source* parameter contains the data block you want to search. The *matchStart* parameter contains the initial match data. Once found, the function accumulates data until the *matchEnd* parameter is found. Matches do not take case into account. This function returns a data object with the matched contents.

The *includeStart* and *includeEnd* parameters tell this function whether to include the starting and ending match data in the returned string. These parameters are set to false by default and may be omitted.

dictionary

The dictionary function builds and returns a dictionary object from the specified parameters. The syntax of this function is as follows:

```
dictionary dictionary (...)
```

Use this function to build a dictionary of key/value pairs in an XQuery function.

dictionary-get

The `dictionary-get` function returns the value for the specified key in a dictionary. The syntax of this function is as follows:

```
node dictionary-get(source, key)
```

The *source* parameter contains the dictionary. The *key* parameter contains the key to search for in the dictionary.

encoded-data-to-string

The `encoded-data-to-string` function converts a data object to a string. The syntax for this function is as follows:

```
string encoded-data-to-string(source, charset)
```

The *source* parameter contains the data object to convert. The *charset* parameter contains the name of the IANA registry character set to use for the conversion.

eval

The `eval` function evaluates the specified XQuery string and returns the result. The syntax for this function is as follows:

```
type eval(string)
```

The *string* parameter contains an XQuery statement to be evaluated.

The `eval` function optionally takes a second argument. In this case, *string* can contain variables that are defined in the second argument, which is a dictionary of key-value pairs. For example:

```
eval("concat('eval may take ',$number, ' argument',$plural)",
dictionary(("number", 2),("plural","s")))
```

This call returns the string “eval may take 2 arguments”.

http-get

The `http-get` function initiates an HTTP GET request for the given URL and returns the results. The syntax for this function is as follows:

```
dictionary http-get(source, headers)
```

The *source* parameter contains the URL string to request. The *headers* parameter is an optional dictionary you can use to specify HTTP request header information. On return, this function returns a dictionary with the keys listed in Table 33.

Table 33 Keys returned by http-get

Key	Description
STATUS_CODE	Contains an integer with the HTTP status code of the request. For example, a value of 404 means the specified URL was not found.
REQUEST_URL	Contains the original URL that was requested.
ACTUAL_URL	Contains the actual URL from which the response came. This value may be different than the requested URL if the request was redirected.
DATA	Contains the data from the request if there were no errors.
ERROR_DATA	Contains the data of the request if there was an error.
HEADERS	Contains a dictionary with the HTTP headers returned by the server.
HEADER_STRING	Contains a string with the HTTP headers returned by the server.

http-head

The `http-head` function initiates an HTTP HEAD request and returns only the header portion. The syntax for this function is as follows:

```
dictionary http-head(source, headers)
```

The *source* parameter contains the URL string to request. The *headers* parameter is an optional dictionary you can use to specify HTTP request header information. On return, this function returns a dictionary with the keys listed in Table 33.

Table 34 Keys returned by http-head

Key	Description
STATUS_CODE	Contains an integer with the HTTP status code of the request. For example, a value of 404 means the specified URL was not found.
REQUEST_URL	Contains the original URL that was requested.
ACTUAL_URL	Contains the actual URL from which the response came. This value may be different than the requested URL if the request was redirected.
HEADERS	Contains a dictionary with the HTTP headers returned by the server.
HEADER_STRING	Contains a string with the HTTP headers returned by the server.

http-post

The `http-post` function initiates an HTTP POST request for the given URL and returns the results. The syntax for this function is as follows:

```
dictionary http-post(source, postdata, headers)
```

The *source* parameter contains the URL string to request. The *headers* parameter is an optional dictionary you can use to specify HTTP request header information. On return, this function returns a dictionary with the keys listed in Table 33.

Table 35 Keys returned by http-post

Key	Description
STATUS_CODE	Contains an integer with the HTTP status code of the request. For example, a value of 404 means the specified URL was not found.
REQUEST_URL	Contains the original URL that was requested.
ACTUAL_URL	Contains the actual URL from which the response came. This value may be different than the requested URL if the request was redirected.
DATA	Contains the data from the request if there were no errors.
ERROR_DATA	Contains the data of the request if there was an error.
HEADERS	Contains a dictionary with the HTTP headers returned by the server.
HEADER_STRING	Contains a string with the HTTP headers returned by the server.

http-request

The `http-request` function initiates a generic HTTP request for the given URL and returns the results. The syntax for this function is as follows:

```
dictionary http-request(url, additionalInfo)
```

You can use this function to initiate a request of any type: GET, POST, HEAD. The *url* parameter contains the URL string to request. The *additionalInfo* parameter contains a dictionary with information to add to the request. This dictionary contains the keys shown in Table 36.

Table 36 Keys for additionalInfo parameter

Key	Description
Method	Contains a string with one of the following values: GET, HEAD, or POST. If you do not include this key, the function uses the default value GET.
DisableAlerts	Contains a boolean that when set to true suppresses error dialogs. This value is set to false by default.
DisableCache	Contains a boolean value that, when set, forces Sherlock to perform an HTTP request to retrieve the item, ignoring any cached copies of it.
DisableRedirects	Contains a boolean value that when set to true suppresses automatic HTTP redirects. This value is set to false by default.

Key	Description
`FavorCache`	Contains a boolean value indicating whether Sherlock should return a cached copy of the item, if one exists, without performing a network check for a newer item. If the item is not in the cache, Sherlock performs an HTTP request to get it.
`FavorCacheUpdate`	Contains a boolean value indicating whether Sherlock should return a cached copy of the item, if one exists, without performing an immediate network check for a newer item. If the item is not in the cache, Sherlock performs an HTTP request to get it. If the item is in the cache, Sherlock returns the cached item and then performs an HTTP request in the background to update the cached data.
`Headers`	Contains a dictionary of additional headers to include with the request.
`PostData`	Contains data to send with a `POST` request.

On return, this function returns a dictionary with keys appropriate to the type of request. The possible keys are listed in Table 33.

Table 37 Keys returned by http-request

Key	Description
`STATUS_CODE`	Contains an integer with the HTTP status code of the request. For example, a value of 404 means the specified URL was not found.
`REQUEST_URL`	Contains the original URL that was requested.
`ACTUAL_URL`	Contains the actual URL from which the response came. This value may be different than the requested URL if the request was redirected.
`DATA`	Contains the data from the request if there were no errors.
`ERROR_DATA`	Contains the data of the request if there was an error.
`HEADERS`	Contains a dictionary with the HTTP headers returned by the server.
`HEADER_STRING`	Contains a string with the HTTP headers returned by the server.

load-service

The `load-service` function loads the services from the specified URL and makes them available to your script code. The syntax for this function is as follows:

```
boolean load-service(url)
```

The *`url`* parameter contains the URL of the file containing the service. The function returns a boolean value indicating whether the services were successfully loaded.

localized-resource

The `localized-resource` function returns a localized resource based on the user's current language settings. The syntax for this function is as follows:

```
type localized-resource(key, baseURL, fileName)
```

The *`key`* parameter identifies a key in the `LocalizedResources.plist` file of the language folder matching the user's current settings. The value associated with this key should be the name of the resource file you want to locate.

The *`baseURL`* and *`fileName`* parameters are optional. Use the *`baseURL`* parameter to override the default location for localized resources, which is specified in the channel configuration file. Use the *`fileName`* parameter to specify a property-list file other than `LocalizedResources.plist` in which to look up the desired key.

localized-url

The `localized-url` function returns the location of the localized resource file based on the user's current language settings. The syntax for this function is as follows:

```
type localized-url(fileName, baseURL)
```

The *`fileName`* parameter specifies the name of a localized resource file. Localized copies of the file must be located in the channel's language folders. The *`baseURL`* parameter is optional and is used to override the default base folder to use when searching for localized resources.

msg

The `msg` function prints a message to stderr. The syntax for this function is as follows:

```
type msg(debug_msg)
```

This function returns the message that was printed. Debugging must be enabled before you can view the output messages. To enable debugging, open a Terminal window on your system and define the environment variable `SHERLOCK_DEBUG_CHANNELS` with a value of 1. Once this variable is defined, launch Sherlock from the command line to begin your debugging session.

null

The `null` function returns the null object. The syntax for this function is as follows:

```
null null()
```

You can use the returned value to determine if an XQuery object is valid.

property-list-decode

The `property-list-decode` function converts an XML property list into a property-list data object. The syntax for this function is as follows:

```
data property-list-decode(source)
```

The *`source`* parameter contains the raw bytes from the XML property-list file. If a parsing error occurs, this function returns an empty object.

property-list-encode

The `property-list-encode` function converts a source object into a data object whose contents are an XML property list. The syntax for this function is as follows:

```
data property-list-encode(source)
```

The *`source`* parameter contains the property-list data you want to encode as XML. The resulting data is appropriate for writing out to an XML file, but is returned as a data object instead of a string.

reg-exp

The reg function performs a pattern match on the specified string and returns the matching text. The syntax of this function is as follows:

```
seq reg-exp(source, reg-exp)
```

The *`source`* parameter contains the string to search. The *`reg-exp`* parameter contains the regular expression to apply to the string. This function returns a sequence of strings, each of which contains the text from a specific match.

The syntax for regular expressions is identical to those for the `matches` function defined in XQuery. For a complete specification of the expression syntax, see `http://www.w3.org/TR/xmlschema-2/#regexs`.

sherlock-function

The `sherlock-function` function calls a Sherlock Web Service to perform a specific task. The syntax for this function is as follows:

```
type sherlock-function(sourceURL, functionName, ...)
```

The *`sourceURL`* parameter specifies the URL for the file containing the web services code you want to access. The *`functionName`* parameter contains the name of a specific function you want to access. If the web-service function takes any parameters of its own, you can pass those in as extra arguments to `sherlock-function`.

source

The `source` function returns an XML string representing the specified node. The syntax for this function is as follows:

```
string source(node)
```

string-combine

The `string-combine` function concatenates a sequence of strings together using a designated separator string. The syntax of this function is as follows:

```
string string-combine(strings, separator)
```

The *`strings`* parameter contains the sequence of strings to be concatenated. The *`separator`* parameter contains the separator string. The function returns the resulting string.

string-separate

The `string-separate` function splits a string based on the specified separator string. The syntax of this function is as follows:

```
seq string-separate(source, separator)
```

The *`source`* parameter contains the string you want to split. The *`separator`* parameter contains the separator string. The function returns the resulting sequence of strings. This function does not include the separator string in the strings of the returned sequence.

string-to-encoded-data

The `string-to-encoded-data` function converts a string to a data object. The syntax for this function is as follows:

```
data string-to-encoded-data(source, charset)
```

The *`source`* parameter contains the string to convert. The *`charset`* parameter contains the name of the IANA registry character set to use for the conversion.

For example, the following function call would yield a data object with the contents `0xffef0031`:

```
string-to-encoded-data("1", "UTF-16")
```

unique-id

The `unique-id` function returns a unique number for the current channel. The syntax for this function is as follows:

```
number unique-id()
```

url

The `url` function returns a CFURLRef object for the specified source. The syntax for this function is as follows:

```
url url(source)
```

The *source* parameter contains a string with the URL text.

url-decode

The `url-decode` function decodes a string with encoded characters, returning a regular text string. The syntax for this function is as follows:

```
string url-decode(urlSource, charactersToLeaveEscaped)
```

The *urlSource* parameter contains the string you want to decode. The *charactersToLeaveEscaped* parameter contain specific characters to leave untouched. The *encoding* parameter specifies the encoding format for the returned string.

The following example decodes the colon and forward-slash encoded characters in the string. The string returned by this function call is "`http://`":

```
url-decode("http%3A%2F%2F", "")
```

url-encode

The `url-encode` function encodes the characters of a URL string. The syntax for this function is as follows:

```
string url-encode(urlSource, charactersToLeaveUnescaped,
        legalURLCharactersToBeEscaped, encoding)
```

The *urlSource* parameter contains the URL string you want to encode. The *charactersToLeaveUnescaped* and *legalURLCharactersToBeEscaped* parameters contain specific characters to modify or leave untouched. The *encoding* parameter specifies the encoding format for the returned string.

The following example encodes the colon and forward-slash characters in the given string. The string returned by this function call is "`http%3A%2F%2F`":

```
url-encode("http://", "", ":/", "UTF-8")
```

url-host

The `url-host` function returns the hostname information from a URL string. The syntax for this function is as follows:

```
string url-host(urlSource)
```

The *urlSource* parameter contains the URL path whose hostname you want to obtain. The return value omits any scheme and pathname information. For example, the following function call returns the string "`www.apple.com`":

```
url-host("http://www.apple.com/index.html")
```

url-last-path-component

The `url-last-path-component` function returns the last path component of a URL string. The syntax for this function is as follows:

```
string url-last-path-component(urlSource)
```

The *`urlSource`* parameter contains the URL whose path information you want to obtain. The return value omits any scheme, hostname, and leading path information. For example, the following function call returns the string "`index.html`":

```
url-last-path-component("http://www.apple.com/test/index.html")
```

url-path

The `url-path` function returns the relative path information from a URL string. The syntax for this function is as follows:

```
string url-path(urlSource)
```

The *`urlSource`* parameter contains the URL whose path information you want to obtain. The return value omits any scheme and hostname information. For example, the following function call returns the string "`/test/index.html`":

```
url-path("http://www.apple.com/test/index.html")
```

url-query

The `url-query` function returns the query attributes of a URL string. The syntax for this function is as follows:

```
string url-query(urlSource)
```

The *`urlSource`* parameter contains the URL whose attribute information you want to obtain. The return value omits any scheme, hostname, and leading path information. For example, the following function call returns the string "`action=search`":

```
url-query("http://www.apple.com/index.html?action=search")
```

url-query-value

The `url-query-value` function extracts the value of an attribute from a URL string. The syntax for this function is as follows:

```
string url-query-value(urlSource, key)
```

The *urlSource* parameter contains the URL whose attribute information you want to obtain. The *key* parameter contains the name of the attribute to return. The return value omits any scheme, hostname, and leading path information. For example, the following function call returns the string "search":

```
url-query-value("http://www.apple.com/index.html?action=search", "action")
```

url-scheme

The `url-scheme` function returns the scheme type of a URL string. The syntax for this function is as follows:

```
string url-scheme(urlSource)
```

The *urlSource* parameter contains the URL path whose scheme you want to obtain. The return value omits any host and pathname information. For example, the following function call returns the string "http":

```
url-scheme("http://www.apple.com/index.html")
```

url-with-base

The `url-with-base` function returns a CFURLRef object built from the specified base and relative paths. The syntax for this function is as follows:

```
url url-with-base(urlSource, base)
```

The *urlSource* parameter contains a relative pathname to append to the URL path in the *base* parameter.

version

The `version` function returns a CFNumberRef with the version of the XQuery implementation currently in use. The syntax for this function is as follows:

```
number version()
```

Creating a New Channel

This article takes you through the steps for creating a simple channel that displays an interface and responds to user input. The goal is to get the channel up and running quickly so that you can see how the user interface and underlying script code interact. The channel in this article does not attempt to connect to the Internet or gather information.

Installing the Sherlock Tools

Before you can begin developing Sherlock channels, you must make sure your system has the proper tools. Sherlock requires the developer tools available for Mac OS X 10.2 or later. In particular, you must have the Interface Builder application to design your channel's interface. You must also have the Sherlock palette before you can add path information to the views and controls of your channel.

Install the developer tools for your system using either the Developer Tools CD that came with Mac OS X 10.2 or by downloading the tools from the developer section of Apple's website. Once you have these tools installed, download the Sherlock SDK from the developer section of Apple's website and install it. This SDK installs a Sherlock palette, documentation, and channel templates for you to use in creating channels.

Create the Channel Project

The Sherlock SDK installs a Project Builder template you can use to create new Sherlock channels. This template implements a simple Internet channel that lets the user enter a search string and retrieve the results. You should use this template as the starting point for any new channels you want to create.

1. Launch Project Builder.
2. Select New Project from the File menu.
3. From the New Project window, select Sherlock Channel from the Standard Apple Plug-ins section.
4. Click Next.
5. Enter the name and location of your project.
6. Click Finish.

Your new project includes a channel configuration file, a main code file, a default icon, and a nib file with a predefined window. The project also includes a Read Me file with the latest instructions on how to modify and debug your channel. You should read these instructions before deploying your channel for testing.

Loading the Channel

When you are ready to test your channel, you need to post it to a web server and tell Sherlock to load it. The simplest way to do this is to enable Personal Web Sharing on your local machine and place your channel files in the Sites directory of an active user. You can then load the channel from a browser address line.

The following steps show you how to load the default channel project. If you added any files to the project, you may need to copy those files to the Sites folder along with the standard channel files that came with the project:

1. In the Sharing System Preference, enable Personal Web Sharing on your test machine.
2. Copy the `SherlockChannel.xml` and `Channel.icns` files and the `Channel` directory to the `Sites` directory of your user home directory.
3. Launch Safari.
4. In the browser address bar, enter the path to the `SherlockChannel.xml` file of your channel. For example, if you put your channel in local user Steve's `Sites` directory, your URL might look like the following:

```
sherlock://localhost/~steve/SherlockChannel.xml
```

Important: Starting with Mac OS X 10.2.4, Sherlock can access your computer (localhost) only when the `SHERLOCK_DEBUG_CHANNELS` environment variable is set to 1. (This requirement may be removed in future versions of Mac OS X). This variable is set for you automatically when you create a new Sherlock project. See "Debug Menu" (page 75) for more information on setting environment variables.

Entering the URL for your channel configuration file launches Sherlock and tells it to load your channel. However, the icon for your channel does not appear in the toolbar until you explicitly add it, either programmatically or from the Sherlock application.

To add a channel to the toolbar from the Sherlock application, you would simply select the Add Channel to Toolbar command from the Channel menu. To add your test channel to the toolbar programmatically, you would use the following URL instead of the one in the preceding step:

```
sherlock://localhost/~steve/SherlockChannel.xml?action=add
```

For more information on loading channels using the `sherlock` scheme, see "Accessing Channels" (page 77).

Debugging Tools

Apple provides several tools to aid you in debugging your trigger code.

Channel Tools

Developers can add the Channel Development Tools subscription to Sherlock to access some special channels: an XQuery code tester, a JavaScript code tester, an HTML View tool, and a XPath Finder tool. The XQuery and JavaScript code testers let you execute trigger code in a Sherlock environment and view the results. The HTML View tool lets you load and view an HTML page. The XPath Finder tool gives you the XQuery path to a specific HTML tag in a document.

To subscribe to the developer channels, go to the following URL from your browser. This URL will load the developer channels and add them to your current subscription list:

```
sherlock://si.info.apple.com/sherlock3s/
                    developerChannels.xml?action=subscribe
```

Debug Menu

Sherlock includes support for runtime debugging of your channel using the Sherlock Debug menu. You enable this feature by launching Sherlock from the Terminal application in the following way:

1. Launch the Terminal application.
2. In your shell, set the environment variable `DEBUG_SHERLOCK_CHANNELS` to the value 1. For example, in the `tcsh` shell, you would use the following command:

   ```
   setenv DEBUG_SHERLOCK_CHANNELS 1
   ```

3. Launch Sherlock from the command-line. You can do this by navigating to the directory containing the Sherlock application bundle and executing the following command:

   ```
   ./Sherlock.app/Contents/MacOS/Sherlock
   ```

Another way to enable the Debug menu is to modify the `SherlockDebug` preference in the system defaults database. When set to `1`, Sherlock displays the Debug menu; when set to `0`, Sherlock hides it. The following example shows you how to enable the Debug menu from Terminal:

```
defaults write com.apple.Sherlock SherlockDebug 1
```

Launching Sherlock with either of these options adds a Debug menu to the end of the Sherlock menu bar. This menu contains commands for reloading the channel and for examining both the data store and the view hierarchy.

The Data Store command displays a browser window with which you can navigate the data store variable space. Following the path hierarchies, you can examine the data currently being managed by your channel at a specific path. You can use this tool to validate the integrity of your channel's data.

The Examine View Hierarchy command displays a window with an outline view that shows the object hierarchy of your channel's views. Each object is identified by its type and by the current address of the object itself. This information is read-only and cannot be permanently changed.

Accessing Channels

Once you have created a channel, you need to test it and make it available for clients to use. The way you test and deploy a channel is over the web. By posting the files for your channel on a local web server, you can access them from the Sherlock application and verify that everything works. When you're ready to deploy, you provide links to your channel using the `sherlock` web scheme.

If you are developing multiple related channels, you can group your channels together and distribute them as a subscription. Subscriptions simplify the process of loading multiple channels by making it possible to load all of the channels with one URL. The subscription itself simply contains a pointer to the individual channels you want to load.

Loading a Channel From a URL

Once your channel is deployed, you access it by entering the URL of your channel configuration file. In order to launch your channel in the Sherlock application, your URL must use the `sherlock` web scheme, that is, it must use the string `sherlock://` in place of the `http://` in the URL. Thus, the URL for accessing a test channel might look something like the following:

```
sherlock://localhost/~steve/MyChannel.xml
```

This URL tells the browser to let Sherlock handle the URL. Sherlock locates your channel configuration file and uses the information in that file to load your channel and display it in the Sherlock application window.

In addition to simply displaying your channel, you can add an `action` attribute to a sherlock URL to perform additional actions. Table 1 lists the actions you can perform on a channel. If you do not specify an action, Sherlock performs the `display` action by default.

Table 1 Supported actions for Sherlock URLs

Action	Description
`add`	Adds the specified channel to the current user's preferences. This action also adds the Sherlock toolbar and menu for the current user. This action also displays the channel in the current window, or in a new window if the `new_window` attribute is present.
`display`	Displays the specified channel in the frontmost window, or in a new window if the `new_window` attribute is present.
`subscribe`	Subscribes the user to the channels in the subscription file specified by the URL.
`toolbar`	Set the value of this attribute to `hidden` if you want to hide the Sherlock toolbar on the window. Set it to `shown` to display the toolbar. If you do not specify this attribute, Sherlock shows or hides the toolbar based on the last user modification. Thus, if the user last hid the toolbar on a window, Sherlock hides the toolbar on the window being loaded.

When adding or displaying channels, you can append the `new_window` attribute to the URL to open the channel in a new Sherlock window. For example, if you want to add your test channel to the toolbar and display the channel in a new window, you would use a URL similar to the following:

```
sherlock://localhost/~steve/MyChannel.xml?action=add&new_window
```

Setting Up Subscriptions

Sherlock supports the ability to subscribe to a group of channels all at once. Subscriptions are convenient if you have several related channels that you want users to install. Rather than force the user to install each channel separately, you can give them the URL of a subscription file and let them install the entire group of channels.

Another advantage of subscriptions is that you can adjust the contents of the subscription file at any time to change the currently available channels. Because the subscription file is located on the network, any changes you make to the file are reflected the next time the user launches Sherlock.

Note: All of the channels in a subscription must have the same hostname. Sherlock enforces this rule and does not load subscriptions containing channels from multiple hosts.

Users can hide and show channels belonging to a subscription using the Sherlock application. The Channels view provides options for moving subscriptions to a folder or toolbar. You can also use this view to unsubscribe to channels altogether.

The following listing shows the format of a channel subscription file and is taken from the developer channels subscription provided by Apple. Each `<channel>` tag contains a URL to a channel configuration file. The `<localized-strings>` tag is used primarily for the localization of the channel name itself. The name you specify is displayed in the Sherlock application preferences window.

```
<channels name="Developer">
    <channel url="channels/xquery.xml"/>
    <channel url="channels/javascript.xml"/>
    <channel url="channels/htmlview.xml"/>
    <localized-strings>
        <localized-string language="en" key="Developer"
                string="Apple Developer Channels" />
    </localized-strings>
</channels >
```

For more information on the syntax for the `channels` and `channel` tags, see "Channels Tags Syntax" (page 34).

Printing Your Channel's Content

This article describes Sherlock's printing support for channels. Sherlock provides both default and custom printing support for channels. With the default support, Sherlock sends a snapshot of your channel's interface to the printer when the user selects the Print command. Default printing happens automatically and requires no effort on your part.

If you want to alter the appearance of your channel prior to printing, you must add code to implement custom printing. With custom printing, you can make minor changes to your channel's existing interface or use a completely separate view to display your data.

This article covers the steps needed to support custom printing in your channel.

Supporting Custom Printing

Custom printing lets you adjust the content and display of your channel prior to it being rendered for the printer. You do not need to implement custom printing support if you want to display the existing contents of your channel without modification. However, if you want to make modifications to content or use a different view for printing, you must enable custom printing.

To enable custom printing, you must tell Sherlock that your channel supports it. To do this, you assign a non-zero value to the `customPrint` path in the data store. You should do this in your channel's initialization code, as shown in the following example:

```
<initialize language="JavaScript">
    DataStore.Set("customPrint", 1);
</initialize>
```

Assigning a value to `customPrint` tells Sherlock that your channel defines a trigger to handle printing. The trigger you define must respond to the `print` path, which Sherlock notifies when it receives a print request. In your trigger, you can adjust the content or print specific portions of your channel. For example, if you want to print the contents of only one subview, you can use your print trigger to reroute the print notification to that subview. The following example reroutes the print notification to the search results table of the channel to print only the search results.

```
<trigger language="JavaScript" path="print">
    /* Print only the results table. */
    DataStore.Notify("MyChannel.ResultsTable.print");
</trigger>
```

If you plan to make significant changes to a channel's printed content, or if you want to rearrange controls in your main view, do not try to use the print trigger alone. For significant printing changes, it is better to define a custom print view to handle printing. For information on custom print views, see "Using a Custom Print View" (page 80).

Using a Custom Print View

If you want to present a different interface for printing than you do for general display, you can do it using a custom print view. A custom print view is another view you create using Interface Builder and store in your channel's nib file. Sherlock loads this view with the rest of the nib file and calls your channel's print trigger at print time to prepare the view for printing.

A custom print view must be an entirely separate view hierarchy from your channel's main display view. Your print view should contain only those elements you want to be printed. You do not have to have a one-to-one correspondence between controls in your display view and print views. You can even use completely different controls in the two views as long as you know how to map information from one view to the other in your print trigger.

When you supply additional views in your nib file, such as a print view, you must tell Sherlock which view to use for displaying content. Sherlock loads all of the views in your nib file by default. If only one view is present, it uses that view for both display and printing. However, if multiple views are present, you must tell Sherlock which view is your main display view. You do this by setting the `mainViewIdentifier` path to the name of your display view during initialization, as shown in the following example:

```
<initialize language="JavaScript">
    /* Indicate which view is for display */
    DataStore.Set("mainViewIdentifier", "MyDisplayView");

    /* Enable printing */
    DataStore.Set("customPrint", 1);
</initialize>
```

Important: Setting the main display view is required if your nib file contains multiple views. Without this information, Sherlock may not display your channel properly.

In your print trigger, you can decide which view to use for printing. If you want to print from a custom view, your print trigger is responsible for populating that view with data prior to printing. When the print command is received, your channel's main display view contains the current data to be printed. Transfer whatever data you need and then send the print notification to your custom view to begin printing.

The following example shows a complete print trigger from the Yellow Pages channel. In this example, the channel copies the selected address, driving directions, and map data over to the view named `PrintView`. Once all the data is set up, the trigger sends a notification to the path `PrintView.print` to begin printing.

```
 <trigger language="JavaScript" path="print">
/* set up the print view */
selectedData = DataStore.Get("YellowPages.data.SelectedRow");
if (selectedData == null)
{
    DataStore.Set("PrintView.name.stringValue", "");
    DataStore.Set("PrintView.address.stringValue", "");
    DataStore.Set("PrintView.phone.stringValue", "");
}
else
{
    DataStore.Set("PrintView.name.stringValue", selectedData.name);
    DataStore.Set("PrintView.address.stringValue", selectedData.address);
    DataStore.Set("PrintView.phone.stringValue", selectedData.phone);
```

```
}

mapImage = DataStore.Get("YellowPages.MapImage.data");
DataStore.Set("PrintView.MapImage.data", mapImage);

drivingDirections =
        DataStore.Get("YellowPages.DrivingDirectionsTable.dataValue");
DataStore.Set("PrintView.DrivingDirectionsTable.dataValue",
        drivingDirections);

/* print */
DataStore.Notify("PrintView.print");
</trigger>
```

Using Web Services

The following tasks show you how to create and use web services in your channel. Web services are remote functions that clients call over an intranet or the Internet to perform specific tasks. For example, the publisher of a stock website can define a service that takes a stock symbol and returns the current price of that stock. Sherlock also defines a type of web service that you link to from your channel code to take advantage of its functionality. You can use existing web services or you can define your own web services to separate out code from your channel that you want to share.

Defining a New Web Service

To publish a new web service, you must first assemble the code file containing your web service code. You place your web service code in an XML file. Because web services contain script code, you use the `<scripts>` tag as the top-level element wrapping your code. When declaring web services, you do not need to include any attributes for this tag. However, when you want to load a web service, you should include a `src` attribute to specify the location of the web service code file.

Inside the `<scripts>` tag, put a `<script>` tag and include the language attribute identifying the type for your functions. The content of the `<script>` tag is the functions you want to define. You can include multiple script tags if your service includes both JavaScript and XQuery syntax.

The following listing illustrates one way to write a web service for finding city, state, and zip code information. The service defines two functions using the XQuery language.

```
<!-- Copyright (c) 2002, Apple Computer, Inc. -->
<!-- All rights reserved.                     -->

<scripts>
<script language="XQuery">

{-- CityStateZipFromZip -  return city/state/zip --}
define function CityStateZipFromZip($zip)
{
    let $query_url := "http://www.usps.gov/cgi-bin/zip4/ctystzip2"
    let $post_data := concat("ctystzip=", $zip)/url-encode(., " ", "+")/
                translate(., " ", "+")
    let $start := "----------&lt;BR&gt;"
    let $end := "ACCEPTABLE"
    let $computedCityAndState := http-post($query_url, $post_data)/DATA/
                data-match(., $start, $end)/normalize-space()
    let $computedCityAndStateArray := string-separate($computedCityAndState,
                " ")
    let $computedState := item-at($computedCityAndStateArray,
                count($computedCityAndStateArray))
    let $computedCity := substring-before($computedCityAndState,
                $computedState)/normalize-space()
    let $newCity := if ($computedCity) then $computedCity else $city
```

```
    let $newState := if ($computedState) then $computedState else $state
    return dictionary(
        ("city", $newCity),
        ("state", $newState),
        ("zip", $zip))
}

{-- CityStateZipFromCityState --}
define function CityStateZipFromCityState($city, $state)
{
    let $query_url := "http://www.usps.gov/cgi-bin/zip4/ctystzip2"
    let $post_data := concat("ctystzip=", $city, " ", $state)/
                url-encode(., " ", "+")/translate(., " ", "+")
    let $computedZip :=  http-post($query_url, $post_data)/DATA/
                data-match(., "&lt;BR&gt;&lt;BR&gt;", "ACCEPTABLE")/
                normalize-space()
    let $firstComputedZip := string-separate($computedZip, " ")[1]
    let $newZip := if ($firstComputedZip) then $firstComputedZip else $zip
    return dictionary(
        ("city", $city),
        ("state", $state),
        ("zip", $newZip))
}
</script>
</scripts>
```

Making your services available is a simple process of placing your web service code file on your web server and notifying clients where they can find it. If you are going to make your web services available to the public, you should also document the syntax for your functions. Clients can then include the code in their scripts using the `<scripts>` tag.

Accessing SOAP Services

In addition to defining your own services, you can also use Sherlock to access other web services using the Simple Object Access Protocol (SOAP). To generate a SOAP request, you must construct the XML query required by the target server. You can then send the request using the Apple-provided functions `http-post` and `http-request`.

The following example, written in XQuery, shows you how to access a SOAP web service for retrieving stock prices. In this example, the `StockSymbolLookup` function creates the XML objects to be passed to the SOAP server as part of the request. The data is then passed to the SOAPQuery method, which creates a set of default HTTP headers, posts the request, and returns the result.

Important: To make the following example more readable, the characters '<' and '>' are not escaped. When writing a channel, however, you must escape these characters to "<" and ">", respectively.

```
{-- SOAPQuery: executes the query --}
define function SOAPQuery($query, $action, $soapAddress)
{
    {-- set the content type and the soap action for the request --}
    let $headers := dictionary (
            ("Content-Type", "text/xml; charset=utf-8"),
```

```
            ("SOAPAction", $action)
        )

    {-- send the request --}
    return http-post($soapAddress, $query, $headers)/DATA
}

{-- StockSymbolLookup based on --}
{-- http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl --}
define function StockSymbolLookup($symbol)
{
    {-- construct the XML query --}
    let $soapQuery :=
    <SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/1999/XMLSchema">
        <SOAP-ENV:Body>
            <ns1:getQuote
                xmlns:ns1="urn:xmethods-delayed-quotes"
                SOAP-ENV:encodingStyle=
                    "http://schemas.xmlsoap.org/soap/encoding/" >
                <symbol xsi:type="xsd:string" > { $symbol } </symbol>
            </ns1:getQuote>
        </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>

    {-- send the request and parse out the result --}
    return SOAPQuery($soapQuery, "urn:xmethods-delayed-quotes#getQuote",
            "http://66.28.98.121:9090/soap")//Result/number()
}
```

To retrieve a stock quote, you would then call the `StockSymbolLookup` function from your code, as shown in the following example:

```
let $stockValue := StockSymbolLookup("AAPL")
```

Trigger Examples

This article includes some sample triggers from the Yellow Pages channel that comes with Sherlock. These examples use a combination of JavaScript and XQuery code to perform searches for the user.

Initiating a Search Using a URL

The following example shows some code from the Yellow Pages channel that handles URL-based search requests. If the user clicks a Sherlock link in an HTML browser, the browser sends the URL text to Sherlock for processing. By the time a URL reaches a specific trigger, Sherlock has parsed the URL text and determined which trigger to call.

In this example, the trigger responds to the path `URL.complete`, which Sherlock calls after it has parsed the URL attributes. The trigger expects the URL request to contain some other attributes and values, in this case a query string and either a `zip` attribute or a `city` and `state` attribute. It populates the data store with these values and then initiates the query. Because the same data store fields are used to store values entered into the controls of the channel's interface by the user, the behavior is the same as if the user had entered the values and clicked the search button.

```
 <trigger language="JavaScript" path="URL.complete">
/* this trigger handles sherlock url query */
query = DataStore.Get("URL.query");
DataStore.Set("YellowPages.MainQueryField.objectValue", query);

zip = DataStore.Get("URL.zip");
if (zip)
{
    DataStore.Set("YellowPages.CityStateZipField.objectValue", zip);
}
else
{
    city = DataStore.Get("URL.city");
    state = DataStore.Get("URL.state");
    if (city != null &amp;&amp; state != null)
    {
        cityAndState = city + ", " + state;
        DataStore.Set("YellowPages.CityStateZipField.objectValue",
                cityAndState);
    }
}

DataStore.Notify("YellowPages.SearchButton.action");
</trigger>
```

Initiating a Search From a Button Click

The following example shows how the Yellow Pages channel responds to the user clicking the search button from the channel interface. The code for initiating a search from a URL click also calls this code to begin the search.

The following two triggers show the initial search setup. In the first trigger, clicking the button sends a new notification to perform the actual setup. The second trigger uses XQuery code to return a dictionary of paths and values. Upon return from the second trigger, Sherlock assigns each value to the path specified by its key, generating appropriate notifications for each assignment.

```
 <trigger language="JavaScript" path="YellowPages.SearchButton.action">
DataStore.Notify("YellowPages.action.yellowPagesSearchSetUp");
</trigger>

<trigger language="XQuery" path="YellowPages.action.yellowPagesSearchSetUp"
            notify="YellowPages.action.yellowPagesSearch">
dictionary(
    ("YellowPages.ResultsTable.dataValue", ()),
    ("YellowPages.ResultsTable.selectedRows", -1),
    ("YellowPages.NetworkArrows.animating", true())
)
</trigger>
```

Once the search parameters are set up, the final notification sent from the `yellowPagesSearchSetup` trigger is to the `YellowPages.action.yellowPagesSearch` path to perform the search. The trigger that responds to this path is shown in the following code listing. The `yellowPagesSearch` trigger performs the search using the web service `YellowPagesSearch` and puts a subset of the results in the channel's results table. It then sets several other properties, storing old search data and keeping track of the current location in the search result set. The `yellowPagesSearchComplete` trigger performs some late housekeeping tasks to indicate to the user that the search is complete.

```
 <trigger language="XQuery" path="YellowPages.action.yellowPagesSearch"
            inputs="csz=YellowPages.CityStateZipField.objectValue,
            query=YellowPages.MainQueryField.objectValue,
            lastCsz=YellowPages.data.lastCityStateZip,
            lastQuery=YellowPages.data.lastQuery,
            moreString=YellowPages.clickAgainString,
            attemptNum=YellowPages.data.attemptNumber,
            lastResults=YellowPages.data.lastResults">
let $attemptNum := if (not($attemptNum)) then 0 else $attemptNum

let $newQuery := if (($query != $lastQuery) or ($csz != $lastCsz) or
            ($attemptNum = 0)) then 1 else 0
let $cityStateZip := string-replace($csz, "/", ", ")
let $resultsToSave := if ($newQuery) then YellowPagesSearch($cityStateZip,
            $query) else $lastResults

let $resultsCount := count($resultsToSave)
let $newResultsStart := 1+($attemptNum * 10)
let $resultsEndMax := $newResultsStart + 9
let $resultsEnd := if ($resultsCount &lt; $resultsEndMax) then $resultsCount
            else $resultsEndMax

let $resultsToShow := sublist($resultsToSave, 1, $resultsEnd)
```

```
let $moreResults := if ($resultsCount &gt; $resultsEnd) then 1 else 0
let $labelText := if ($moreResults) then $moreString else " "
let $attempt := if ($moreResults) then $attemptNum+1 else 0

return dictionary(
    ("YellowPages.ResultsTable.dataValue", $resultsToShow),
    ("YellowPages.data.lastResults", $resultsToSave),
    ("YellowPages.action.yellowPagesSearchComplete", unique-id()),
    ("YellowPages.ResultsTable.highlightedColumn", "distance"),
    ("YellowPages.ResultsTable.columns.distance.indicatorImage",
            "../shared/Ascending.tiff"),
    ("YellowPages.data.sortColumn", "distance"),
    ("YellowPages.data.sortOrder", "ascending"),
    ("YellowPages.ClickAgainText.stringValue", $labelText),
    ("YellowPages.data.lastCityStateZip", $csz),
    ("YellowPages.data.lastQuery", $query),
    ("YellowPages.data.attemptNumber", $attempt)
)
</trigger>

<trigger language="XQuery"
        path="YellowPages.action.yellowPagesSearchComplete">
dictionary(
    ("YellowPages.ResultsTable.selectedRows", 0),
    ("YellowPages.NetworkArrows.animating", false()) )
</trigger>
```

Opening a New Channel From a Trigger

If your channel wants to redirect the user to another channel to handle some task, you can do so programmatically from your channel's controls. To open another channel, simply ask Sherlock to open a URL that is prefaced with the sherlock web scheme identifier (`sherlock://`). When opening a channel, you can specify a URL with the location of the channel's main XML file. For channels that are already registered with Sherlock, you can specify the channel identifier instead of a file URL. The following example opens a new Sherlock window for the Internet channel using the channel identifier. The example uses the `new_window` identifier to open the channel in a new window.

```
<trigger language="JavaScript" path="MyChannel.MyButton.action">
    System.OpenURL("sherlock://com.apple.internet?new_window");
</trigger>
```

Document Revision History

This table describes the changes to *Sherlock Channels*.

Date	Notes
2007-04-09	Sherlock is unsupported in Mac OS X v10.5 and later.
2003-12-02	Minor bug fixes.
2003-09-18	Updated content to include additions for Mac OS X 10.3
	AddressBook object added to JavaScript object reference.
	HTMLView and SherlockAddressComboBox UI objects documented.
	Updated other objects to include new attributes.
	Documented the `help_file` attribute for the `channel_info` tag.
	Updated channel version information.
	Updated document to reflect current Sherlock support for the XQuery specification.
2003-05-01	Updated content for WWDC.
2002-08-23	First version of *Sherlock Channels*.