Cincom

Cincom Smalltalk™ VisualWorks®

Web Server Developer's Guide

VisualWorks 8.3

P46-0153-03

SIMPLIFICATION THROUGH INNOVATION

Notice

Part Number: P46-0153-03

Software Release: 8.3

This document is subject to change without notice.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About this Book

This Guide provides comprehensive instructions for using the SiouX framework. SiouX enables VisualWorks developers to build secure, scalable, high-performance web servers. It provides HTTP and HTTPS support for both new and existing web application frameworks (e.g., AppeX, Seaside, Web Toolkit Servlets), and Web services implemented in VisualWorks.

SiouX is an all-new replacement for the existing VisualWorks Application Server (VWAS), and legacy applications based on Opentalk-Web.

Audience

The discussion in this book presupposes that you have at least a moderate familiarity with object-oriented concepts and the VisualWorks environment. It also presupposes that you have a good understanding of web (HTTP) servers and web application development.

It is also assumed that you have a beginning knowledge of programming in a Smalltalk environment, though not necessarily with VisualWorks.

For introductory-level documentation, you may begin with a set of on-line VisualWorks Tutorials, and for an overview of Smalltalk, the VisualWorks development environment and its application architecture, see the *Application Developer's Guide*.

Conventions

We follow a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks tools.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > **Open…**	Indicates the name of an item (**Open…**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	Select (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks window (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left Button	Left Button	Button
<Operate>	Right Button	Right Button	<Option>+<Select>
<Window>	Middle Button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and personal-use license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support.

Cincom provides all customers with help on product installation. For other issues, send email to: helpna@cincom.com.

Before contacting Technical Support, please be prepared to provide the following information:

- The release number, which is displayed in the Welcome Workspace when you start VisualWorks.
- Any modifications (patch files, auxiliary code, or examples) distributed by Cincom that you have loaded into the image. Choose **Help > About VisualWorks** in the VisualWorks Launcher window. All installed patches can be found in the resulting dialog under **Patches** on the **System** tab.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, use **Copy Stack**, in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to Technical Support.
- The hardware platform, operating system, and other system information you are using.

You can contact Technical Support as follows:

E-mail	Send questions about VisualWorks to: helpna@cincom.com.
Web	Visit: http://supportweb.cincom.com and choose the link to Support.
Telephone	Within North America, call Cincom Technical Support at +1-800-727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time. Outside North America, contact the local authorized reseller of Cincom products.

Personal-Use Licensees

VisualWorks Personal Use License (PUL) is provided "as is," without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main window's menu bar (i.e., the Launcher window), and select one of the help options.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Chapter

1

Overview

Topics

SiouX is a framework for developing secure, extensible, high-performance web servers. It provides HTTP and HTTPS support for both new and existing web application frameworks (e.g., AppeX, Seaside, Web Toolkit Servlets), and Web services implemented in VisualWorks.

SiouX is an all-new replacement for the existing VisualWorks Application Server (VWAS), based on Opentalk-Web.

Design Features

SiouX provides support for:

- High-performance implementation of HTTP/HTTPS
- Multiple ports (multiple listeners), multiple applications at one address
- Multiple protocols (HTTP/1.1, HTTP/2, WebSocket)
- SSL and TLS
- Localization and NLS support
- Session management
- Simple, flexible configuration
- Announcements and multiple log formats
- GUI Tool to create and configure servers
- Backward compatibility

The core SiouX framework is a high-performance HTTP server, that includes optional loadable components to provide secure HTTP, session management, Web socket, and Servlet support.

SiouX supports HTTP/1.1, and HTTP/2 protocol versions, SSL 3.0, TLS 1.0, 1.1 and 1.2. NLS (National Language Support) functionality includes server-side support for locales, specified in HTTP headers, cookies, or the URL. Support for both streaming and chunked responses is included, and the server includes a variety of fortification measures to protect against attacks (e.g., setting the rate at which new connections are accepted, the maximum size of a request status line, limiting the total number of header bytes per request, etc.).

Server configuration may be handled either using files or code pragmas, and support for logging with either industry-standard or custom log files is included.

For backward compatibility with existing applications built using the VisualWorks Application Server (VWAS), an optional servlet API can also be loaded.

SiouX Components

The SiouX framework is comprised of the following components:

SiouX-Server

Core package of the SiouX framework, providing basic HTTP server infrastructure, announcements, and loggers.

SiouX-Server-Secure

Adds SSL/TLS support (i.e., to enable HTTPS connections). May be loaded as optional functionality.

SiouX-Sessions

Provides session support. May be loaded as optional functionality.

SiouX-Http

High-performance HTTP/1.1 implementation using Xtreams. AppeX applications use this implementation.

Protocols-Http

Provide support for the `Upgrade` header field. A server `Listener` class should be configured with protocol versions to which it allows an upgrade. By default, the protocol version is HTTP/1.1 only.

Protocols-Http2

Implements the HTTP/2 protocol.

SiouX-Http2

Provides SiouX server support for the HTTP/2 protocol.

SiouX-Net-Http

Backward-compatible HTTP implementation using the Net framework. This is required for existing Seaside applications and Web Services.

SiouX-Tools

Web Server Configuration tool. May be loaded as optional functionality.

SiouX-WebSocket

Support for WebSockets. May be loaded as optional functionality.

SiouX-Servlet

Backward compatibility support for Servlet applications. May be loaded as optional functionality. Example applications are provided in the SiouX-Servlet-Demo package.

SiouX-Examples

Example code to illustrate the SiouX framework.

Architecture

There are just a few core classes in SiouX framework: Server, Listener, Connection, Responder, RequestContext and RequestFilter:

Server

A Server object has one or more Listener objects listening for incoming connections. Once a connection is established, it waits for incoming requests of a matching type, which are then passed back to the server object to be dispatched to a Responder object for execution.

Typically, a single Server object can have several listeners. Each Listener listens on one port, using Responder objects to dispatch against the incoming URL. SiouX is designed such that multiple instances of class Server may coexist and be active in the same VisualWorks image, as long as their Listener objects are configured to listen on different ports.

While it is active, a Server object generates announcements for events such as opening and closing connections, receiving requests or sending responses. These Announcement objects are used for logging server activity and errors.

For further details, see the discussion of class Server.

Listener

A Listener manages a single entry point identified by a socket (an instance of IPSocketAddress). When the Listener starts, it forks a process to listen for incoming TCP connections, and remains bound to this address. When the Listener accepts a TCP connection, a new Connection instance is created and activated.

The protocol which the Listener accepts (e.g. HTTP/1.1, HTTP/2, etc.), and therefore the type of Connection that gets activated, is specified using a subclass of class ProtocolVersion.

A successfully-opened connection forks a request-handling process in which all requests received via that Connection are handled, and the listening process goes back to waiting for next connection.

A Listener is also notified when one of its connections has closed, which allows it to keep count of all currently-active connections that have been accepted through its socket. To ensure robust performance under adverse conditions, SiouX provides a fortification mechanism to monitor and control the number of new connections.

For further details, see the discussion of class Listener.

Connection

A Connection implements a particular protocol (e.g. HTTP or HTTPS). When a connection is open, a background process is started that waits for incoming requests, passes them to the Server for dispatch to a Responder, and then sends the resulting Response objects back to the client. A connection is closed as dictated by the protocol, or when the Server is shut down, or when certain exceptions or application-specific events occur.

For further details, see the discussion of class Connection.

Responder

When a Server object receives a Request object via a connection, it queries each associated Responder to find one that will accept the request. Generally, a Responder uses the status line of the request (i.e., method, URL path, protocol version) to dispatch it. By default,

a Responder accepts a request whose URL path it matches, but your application can define and use custom subclasses of Responder that specify their own criteria for accepting requests.

A Responder object can be configured with a collection of RequestFilter objects. RequestFilters enable the Server object to process incoming requests in a flexible and configurable manner.

For details, see the discussion of class Responder.

RequestContext

Instances of RequestContext hold all objects related to a specific request: a request, connection and responder. For details, see the discussion of class RequestContext.

RequestFilter

Subclasses of RequestFilter are used to configure Responder instances in a flexible manner. RequestFilter subclasses contain the logic for class SessionFilter, which handles the session management, class LocaleFilter, which handles the extraction of a locale from an incoming request, based upon a set of rules, and class ETagRequestFilter that enables client-side resource caching through the use of the ETag response header.

For details, see the discussion of class RequestFilter.

Compatibility

To offer both high performance and backward compatibility with existing application code, SiouX can be configured to use one of two different protocol implementations (back-ends): SiouX-Http, a new implementation that uses the Xtreams package, and SiouX-Http-Net, which uses the existing VisualWorks Net framework.

The first implementation, SiouX-Http, is used by the AppeX web application framework, and is suitable for building custom RESTful web services. The second, SiouX-Net-Http, uses the Net framework and is a backward-compatible replacement for the older Opentalk-based VisualWorks HTTP server. SiouX-Net-Http is suitable for running

applications written using Seaside, the Web services framework, and Web Toolkit Servlets.

Going forward, the Xtreams-based SiouX-Http provides greater flexibility and performance, for where the Net implementation includes the overhead of pre-parsing all HTTP headers and message bodies, the Xtreams implementation aims for higher performance by parsing HTTP requests as little as possible.

While the newer SiouX-Http back-end offers superior performance, porting existing Seaside and Web Service applications to use Xtreams may involve substantial changes. In this situation, SiouX-Net-Http can be used to run your existing application code, while taking immediate advantage of the other new features of SiouX.

To distinguish between Net and Xtreams implementations, the documentation may at times use their name spaces, e.g.: SiouX.HttpResponse and SiouX.HttpRequest vs. Net.HttpResponse and Net.HttpRequest.

Loading SiouX

SiouX is delivered as a group of support parcels that may be loaded into your VisualWorks development image. These parcels are located in the `/www` subdirectory of the standard VisualWorks distribution.

For developing web services or web applications, SiouX is configured using VisualWorks package prerequisites, such that the correct SiouX parcels are automatically loaded when you load a web services or web application framework. E.g., by loading AppeX, the SiouX-Server and SiouX-Http parcels are loaded as prerequisites.

Alternately, you can begin by loading SiouX itself. To build a SiouX server, for example, you would start by loading the SiouX-Server and SiouX-Http parcels, or SiouX-Server and SiouX-Http-Net, if you wish to later use the legacy web sevices or web application frameworks.

To build a SiouX server that supports both HTTP/1.1 and HTTP/2 protocol versions, load the SiouX-Http2 parcel.

To load SiouX into your VisualWorks image, open the Parcel Manager (select **Tools > Parcel Manager** in the VisualWorks Launcher window), select the suggested **Web Development** extensions, and load the SiouX-Server parcels and any others by double-clicking on the desired items in the upper right-hand list of the Parcel Manager.

Example: "Hello World!"

The traditional "Hello World!" example can illustrate how to quickly create a server using SiouX.

1. To begin, use the Parcel Manager to load the SiouX-Http parcel (located in the **Web Development** category).

   This will automatically load the SiouX-Server parcel.
2. Open a System Browser and create a new responder class, as a subclass of SiouX.HttpResponder.

   For this, select **Class > New Class**... in the Browser, and provide the following details in the **New Class** dialog:

Package	(none)
Name space	Smalltalk.SiouX
Name	Hello
Superclass	SiouX.HttpResponder

3. The newly-created class definition should be:

```
Smalltalk.SiouX defineClass: #Hello
        superclass: #{SiouX.HttpResponder}
        indexedType: #none
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: ''
```

4. Create a new message category named **services** (e.g., select **Protocol > New** ... in the browser, and enter the message category name **services**).
5. In the source pane of the browser, write the following method:

```
executeRequestFor: aRequestContext
```

```
^aRequestContext response
 contentType: 'text/html';
 contents: '<HTML><BODY><H1>Hello World!</H1></BODY></HTML>';
 yourself
```

6. Select **Accept** from the <Operate> menu.
7. In a Workspace, evaluate the following code to create a server and start it:

```
server := Server id: 'HelloServer'.
server addResponder: (responder := Hello new).
responder path: '/hello'.
listener := server listenOn: 8000 for: HttpConnection.
server start.
```

   Sending #path: to set the responder path is optional, but if no path has been specified, the responder consumes all requests to the specified port.
8. Launch a Web browser and open the following URL:

```
http://localhost:8000/hello
```

   The browser should display: **Hello World!**
9. To stop and release the server, evaluate the following Workspace code:

```
server release.
```

Chapter

2

SiouX Framework

Topics

This chapter describes the behavior and APIs of the core classes in the SiouX framework, and offers some code examples to illustrate their usage.

This chapter also presents how to create basic server, listener, responder, and filter configurations using the Web Server Configuration Tool.

The SiouX core classes are: `Server`, `Listener`, `Responder`, `RequestContext`, and `RequestFilter`.

Server

Each SiouX server object (an instance of class Server) has at least one Listener and one Responder object. The listener manages connections with web clients, while the responder dispatches and executes their web requests. A server object may be configured to listen on one, or several ports simultaneously, and SiouX allows multiple server objects to run concurrently in a single VisualWorks image (e.g., HTTP on port 80, and HTTPS on port 443).

A Server object is configured with one or more Responders, where the order in which they are registered with the server dictates their assigned priority. When a request object is passed to a server, each registered responder is offered a chance to handle it, after all higher-priority responders have rejected it. If a responder rejects a service request (by returning nil), the server tries again with the next-lower priority responder. This iteration ends with the first responder that returns a response.

If no Responder accepts a request, the server itself responds with the default response: 404 Not Found. This default may be changed using Server class >> noResponderReply: aBlock.

The active instances of class Server are managed through a global shared variable Server.Registry. When an instance is created it is automatically registered with a unique ID, that is either explicitly specified or auto-generated. Each server's activity is controlled with the #start and #stop messages. When a Server instance is to be discarded, it is sent the message #release, which removes it from the global registry.

Creating a Server

There are three basic ways to create and configure a SiouX server: (1) programmatically, (2) using Server and Responder pragmas and (3) using the Web Server Configuration tool.

Creating a Server using Code

The following example requires the SiouX-Http package, and uses a sample Responder from the SiouX-Examples package.

To begin, create a server instance with the id 'Experiments'. This new instance will be registered in the global Server.Registry under this id:

```
server := SiouX.Server id: 'Experiments'.
```

Create a listener for port 4242 on 'localhost', and use an HttpConnection:

```
server listenOn: 4242 for: SiouX.HttpConnection.
```

The #listenOn:for: method can be used to create several distinct Listener objects on different ports of the same server.

Next, add a Hello responder to the server. Class Hello is a subclass of SiouX.HttpResponder, that is already defined in the package SiouX-Examples.

```
server addResponder: SiouX.Hello new.
```

The Hello responder accepts requests with the first URL path token as 'hello', e.g.:

```
http://localhost:4242/hello
```

This is available via the responder's #path method.

Next, configure the server to use a connection announcement. With this, we can observe the communication between client and server:

```
server
 when: SiouX.ConnectionAnnouncement
 do: [:ann | Transcript cr; print: ann].
```

Start the server, and send a GET request:

```
server start.
['http://localhost:4242/hello' asURI get]
 ensure: [server stop].
```

The VisualWorks Transcript shows a series of announcements:

```
http 127.0.0.1:50029* opening
http 127.0.0.1:50029* GET /hello HTTP/1.1
http 127.0.0.1:50029* prepared HTTP/1.1 200 OK
http 127.0.0.1:50029 closed
```

To finish, remove the server from the global registry:

```
server release.
```

Shutdown a Server after a Delay

Alternately, a server may be stopped after a specified internal using `#stop: aDuration`, where `aDuration` indicates a delay between the shutdown notification and actual shutdown.

Configuring a Server using Method Pragmas

The second way to configure a server is more declarative: using method pragmas. With this approach, the server configuration is a part of the application code. That is, a server and its listeners can be declared with an class method extension to the `SiouX.Server` class, that is adorned with the `#server:` pragma. In this way, the server is configured as soon as you load code into the VisualWorks image.

To illustrate, the `SiouX-Examples` package includes a predefined server identified as `Examples`. The method looks like this:

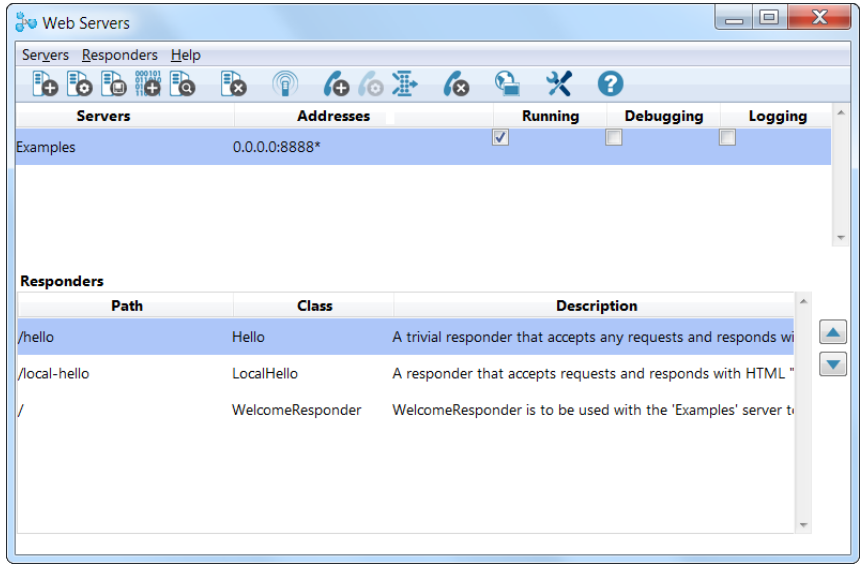
```
Server class>>examples: aServer
 <server: 'Examples'>
 aServer listenOn: 8888 for: HttpConnection
```

Here, `aServer` is configured to listen for HTTP connections on port `8888`. Configuration methods can follow this pattern to receive an instance of `Server` as their single argument, allowing developers to add listeners and/or set additional configuration parameters on the specified server object.

For details, see: Configuring a Listener, and Configuring Responders with Pragmas.

Creating a Server using the Configuration Tool

The Configuration Tool provides a UI for creating and configuring servers. Configurations may be saved to or loaded from XML files. To open the Configuration Tool, select **Tools > Configure Web Servers** from the VisualWorks Launcher.



The Configuration Tool displays the status and basic information for all active servers (above) and selected server responders (below). Three columns in the **Servers** pane provide check boxes to toggle server parameters:

Running

Starts and stops a selected server. This check box becomes active when the specified server has been configured with at least one `Listener`.

Debugging

Enable or disable debugging for the selected server.

Logging

Enable or disable all server logging. The logs are added to the server using the **Server Logs** menu. The box appears checked if at least one log is active.

Servers Menu

Options on the **Servers** menu:

Create…

Create a new server with the specified id. Select the server and finish configuration. To finish configuration you need to provide a Listener (use the **Set Listener** menu item) and Responder (use the **Responders** menu).

Add…

Open a list of preconfigured servers that are configured using pragmas. Select one to activate.

Save…

Save the selected server configuration to an XML file.

Save All…

Save all server configurations to an XML file.

Load…

Load a server (or servers) configuration from an XML file.

Delete

Stop the selected server and remove it from the global registry.

Logs

Add or remove a log from the selected server. The following options are available: CommonLog, StatusLog, or ErrorLog. Use the **Add to Server** check box in the **Logging** dialog, to add (checked) or remove (unchecked) the log.

Remove all Logs

Stop all server logs and remove them from the selected server.

Inspect

Open an inspector on the selected server instance.

Set Listeners

Add a listener or update the server listener options. For details, see: Configuring a Listener.

Working with Certificates and Private Keys

To provide secure HTTPS, SiouX supports SSL/TLS, standard digital certificates (public), and private keys.

Since current web browsers no longer accept self-signed certificates, to set up a secure HTTP server using SiouX, you must first obtain a certificate file from a Certificate Authority (CA), and obtain a private key. For example, Digicert provides a certificate-creation service, as well as details on obtaining a private key.

Note: At this time, SiouX does not support encrypted private keys.

VisualWorks provides a facility to generate self-signed certificates and private keys for testing purposes only. For details, refer to the X509 package comments.

To configure a secure SiouX server, you must provide a file containing a "server certificate chain" that a Listener can use to authenticate the server for clients. Two certificate formats are supported: (1) PEM; and (2) DER encoding.

The textual PEM format contains a single chain of certificates sorted with the subject certificates after their issuer certificates, starting with a self-signed well-known CA certificate, and ending with the actual server certificate. We recommend using this certificate format. Note that in practice the well-known CA certificate can be omitted since it is well known. Most CAs provide two separate files: one containing the certificate chain up to the Root CA, and a second file with the subject certificate itself. The textual format is indicated with a `.PEM` or `.CRT` file extension.

Alternately, the file can contain a binary dump of a single certificate in its standard DER encoding. This format does not support multiple certificates, so it is only suitable for cases in which the server certificate was issued directly by a well-known CA. Typical file extensions used for the binary format are `.DER` or `.CER`, but any filenames with other than `.PEM` or `.CRT` extensions are assumed to be binary.

Configuring a Secure Server

To create a secure HTTP server, first load the SiouX-Server-Secure package. This package adds SSL/TLS support for Connection objects. In summary, to enable HTTPS on a Server, add a Listener for HttpsConnection with the desired address and port (as illustrated below). The Listener must be configured with the proper TLSContext. At the very least, the context must be configured with a valid server certificate and private key. (For details on how to obtain them, see: Working with Certificates and Private Keys.)

Creating the Server and Listener

To set up a secure SiouX server using certificate files, and to configure the server for the HTTP/1.1 or HTTP/2 protocol, we can begin with the following code:

```
server := SiouX.Server id: 'Experiments'.
server addResponder: SiouX.Hello new.
listener := server listenOn: 8000 for: SiouX.HttpsConnection.
```

After receiving a certificate file from a CA and obtaining a private key file, use them to set a secure Listener on the server:

```
server
 setSecureListener: listener
 certificateFile: 'myCertificate.pem'
 privateKeyFile: 'myPrivateKey-rsa.key'.
```

Specifying the Supported HTTP Versions

By default, a SiouX server supports the HTTP/1.1 protocol. If you would like your server to support HTTP/2, add the protocol negotiation extension to the TLS context. I.e., set TLSAppLayerProtocolNegotiation defaults to #(h2 http/1.1), and add the extension as follows:

```
alpn := Xtreams.TLSAppLayerProtocolNegotiation defaults.
listener tlsContext addExtension: alpn.
```

For HTTP/2, the Listener should also be configured with the protocol version which includes the appropriate settings, e.g.:

```
listener protocolVersions: (Array with: Protocol.HTTPv20 new).
```

By default, the listener protocol version is class Protocol.HTTPv11 and doesn't need to be set by your application code. For HTTP/2, there is a convenience method #useHTTP2Protocol that sets up the configuration for the cipher suites, TLS extensions, compression and protocol versions:

```
listener useHTTP2Protocol.
```

Once the server and listener have been configured, start the server:

```
server start.
```

Class Protocols.HTTPv20 includes server settings, which may be modified to tune the server. For details, see: Specifying the HTTP/2 protocol for a Listener.

Testing the Server

To confirm that the server is functional, we can test it with a client.

Note: The VisualWorks 8.3 release includes class HttpClient, which only support HTTP/1.1. A preview version of class HTTP2Client is provided as part of the HTTP2 package in the `/preview` directory. As preview code, this class and its public interface are subject to change. Class HTTP2Client may be used for testing the HTTP/2 server, and provides all functionality to start a connection, as well as to send and receive messages.

For the purposes of this test, the client doesn't need to specify a certificate, so we can use a default TLS context:

```
clientContext := TLSContext newClientWithDefaults.
```

(For details on configuring an HTTP client with certificates, refer to the *Internet Client Developer's Guide*.)

Testing the server, using an HTTP2Client:

```
clientContext
 suites: (TLSCipherSuite suites: #(tls12 (#ecdhe #(#sha256 #sha384)))).
clientContext addExtension: Xtreams.TLSAppLayerProtocolNegotiation defaultALPN.
client := Net.HTTP2Client new.
client protocolVersion: Protocols.HTTPv20 new.
client tlsContext: clientContext.
client tlsSubjectVerifier:
 [:cert | true]. "so that the client doesn't signal TLSBadCertificate"
response := client get 'https://localhost:8000/hello'.
```

Shutting down the Server

To shut down the server:

```
server stop.
```

Don't forget to release the server and certificates when they are no longer needed. This will also release the certificate store and any external resources.

```
certificates release.
clientContext release.
serverContext release.
server release.
```

Specifying Client Authentication

The server can force the clients to authenticate with their own certificates as well. For this, use the tlsVerifier option on the listener object. The parameter to #tlsVerifier: is a block that receives client certificates. This is required to verify that the subjects of those certificates are allowed to connect to the server. For example:

```
listener tlsVerifier:
 [:certificate |
 certificate ifNil:
  ["The client didn't send a certificate.
  Returning true will proceed with handshake.
  Returning false disconnects"
  ^false].
```

```
PermittedClients includes: certificate subject commonName]
```

Here, we assume that the server decides whether to accept an empty certificate list or not. If the verifier returns `true`, we proceed with the handshake without `CertificateVerify`, otherwise we disconnect.

Securing a Server

To understand the proper use of the TLS protocol, it is highly recommended that you study the `TLS` package. Improper configuration can cause serious vulnerabilities in your server.

It is also necessary to configure adequate TLS session caching and resumption, as this has a critical impact on server performance. All TLS-related parameters are configured through the `TLSContext` object. For details, refer to the code comments for the `TLS` package.

Debugging a Server

Since a `Server` interacts with a number of background processes (e.g., for handling announcements), errors are generally suppressed. Nevertheless, since there are times when it may be necessary to debug internal server code, a special flag is provided to enable the debugger to open for exceptions that are raised inside the server implementation proper.

To enable debugging for a server, use the Web Servers Configuration Tool (open this by selecting **Tools > Configure Web Servers** in the Launcher window). Select the server you wish to debug, and click on the **Debugging** checkbox.

Note: Enabling the debugging flag for a Server can sometimes interfere with proper operation of its network protocol; therefore this flag should never be enabled for a deployed server.

Listener

Instances of class `Listener` manage client connections to a server object. Each of the `Listener` objects belonging to a `Server` runs its own process. These listener processes wait for incoming connection requests.

When an incoming connection request is accepted, the listener process then creates a new Connection object, and activates it with the #open message. This spawns a dedicated server process to henceforth manage the newly-established connection.

The connection process reads incoming requests from the connection socket, builds a HTTP request object from it (further parsing some header fields). Finally, it spawns a worker process that "executes" the incoming request.

The behavior of each Listener instance is dictated by a number of parameters (held in its instance variables). The #connectionCount is used to control the rate at which new connections are accepted. When the connection count reaches #lowerConnectionLimit, the listener process begins to introduce a delay in the accept loop, thus slowing the rate of acceptance. As the connection count climbs toward the specified #maxAcceptDelay, the delay gets progressively longer. When the connection count reaches #upperConnectionLimit, the listener process is suspended entirely until the count drops back below the limit. The listener announces crossing these thresholds with several ListenerAnnouncements that can be used to monitor its state.

For some suggestions about how to tune these limits, see the discussion: Fortifying a Listener.

Configuring a Listener

To configure a Listener using the Configuration Tool, select **Servers > Set Listeners** to open a list of all listeners associated with the selected server, and show their basic configuration options. In this dialog, you can add new HTTP or HTTPS listeners, and **Change**... or **Delete** a listener selected from the list.

The basic **Listener Settings** are as follows:

Address

The IP address of the interface to which the server should be bound. If unspecified, the server is bound to all available interfaces.

Binding the server to localhost/127.0.0.1 makes it accessible only from the same host. Changing the address while the server is

running will shut down the server and restart it on the new address.

Port

Specifies the server's port number. Changing the port while the server is running will shut down the server and restart it on the new port number.

Reuse Address

Specifies the `SO_REUSEADDR` option on the listening socket of the server. This is only needed when you expect to restart your server frequently. In order to have an effect, this option must be enabled when the server starts.

If this option is disabled, you may have to wait up to eight minutes after stopping the server until the OS allows you to start the server again on the same port. If that is not an issue though, it is better to disable this option.

HTTPS listeners require both a valid certificate and a private key. For details on how to obtain them, as well as the supported file formats, see: Working with Certificates and Private Keys. The listener settings for these keys are:

Certificate

Name of the file that contains a server certificate. The listener uses this certificate to authenticate the server for connecting clients. The complete certificate chain for this setting can be obtained by concatenating the certificate chain and the subject certificate into a single file.

The certificate(s) are extracted from the file immediately when this setting is applied.

Private Key

Name of a `PKCS8` (PEM-encoded) file that contains a private key corresponding to the server certificate (specified by the **Certificate** setting, described above).

The key is extracted from the file immediately when this setting is applied.

The **Listener Settings** also include load handling and fortification options:

Connection Priority

Priority of the connection process that is created for each established connection.

Tuning this priority (relative to worker process and listener process priorities) allows you to place emphasis on different aspects of the request processing. If it is more important to accept new connections than serve the existing ones, the listener priority should be higher. If you want requests to be processed faster than new ones being received, then worker priority should be increased.

Note that it is very easy to starve other activities if you give too much priority to something that takes a lot of effort. You need to tune the priorities to the particular needs of your specific application, the load patterns that you expect, the amount of effort it takes to complete each request, etc. It may take several attempts to get it right.

When this setting is changed, the new value applies only to new connections established after that point.

Connection Timeout (ms)

Time to wait before terminating an idle connection. The value is in milliseconds.

Listener Priority

Priority of the `Listener` process. By adjusting this priority (relative to server process and worker process priorities) you can emphasize different aspects of request processing.

Listener Backlog

The size of the TCP listening backlog. This determines (depending on how the operating system treats it) the maximum number of incoming socket connections waiting to be accepted.

Lower Connection Limit

The lower threshold on the number of concurrent connections. When the number reaches this value, the server starts considering itself to be getting overloaded. Above this value it will start slowing down the rate at which new connections are accepted by delaying the listener after each accept, ultimately reaching the maximum delay listed in the settings. If the server reaches the maximum number of connections, it will shut off the listener entirely and not resume it until the number of connections drops below this setting. For example if the maximum limit was 100 and we set this to 90, then when we reach 100 simultaneous connections, all new connections would be held until we had closed 10 of the active connections, reaching the lower connection limit of 90.

Upper Connection Limit

The maximum number of simultaneous connections allowed. Note that some operating systems may impose a maximum of 1024 connections, regardless of this setting. The more work your processes are likely to do for each incoming request, the lower you're likely to want this setting.

Maximum Delay (ms)

As the load on the server (the number of concurrent connections) increases above the lower connection limit, the listener begins to delay accepting new connections. This delay increases linearly as the upper connection limit is approached. Note that the resolution of the timers available on the server operating system may affect the actual delays.

Request Status Line Limit

The maximum number of bytes allowed for a status line.

Request Header Size Limit

The maximum number of bytes allowed for a request header.

Specifying the Protocol Version of a Listener

A Listener can be configured to respond to different protocol versions. For this, several subclasses of ProtocolVersion are provided. Currently, the following protocol versions are supported:

Protocols.HTTPv11

The default protocol version. The class knows how to upgrade itself to HTTP/2 or WebSocket.

Protocols.HTTPv20

After upgrading a connection to HTTP/2 version, this class performs a client-server handshake and starts a multiplexer to process all requests.

SiouX.WebSocket

After upgrading a connection to the WebSocket protocol, this class performs a client-server handshake, and opens a WebSocketConnection to send and receive WebSocketMessage messages.

To configure a Listener to accept protocols other than the default, use #protocolVersions: with an Array of subclasses in the ProtocolVersion hierarchy, e.g.:

```
server := Server id: 'SiouXExamples'.
listener := listenOn: 8000 for: HttpConnection.
listener protocolVersions: (Array with: WebSocket new).
```

Using this code, an HTTP 1.1 connection would be created, but a request that includes the "Upgrade: websocket" header field would be upgraded to the Websocket protocol.

Specifying the HTTP/2 protocol for a Listener

By default, the protocol version of a connection is HTTP/1.1. To specify HTTP/2, there are two steps:

1. Specify the protocol in the TLS Application-Layer Protocol Negotiation Extension, e.g.:

```
alpn := Xtreams.TLSAppLayerProtocolNegotiation defaultALPN.
```

By default the preferred protocols are: ('h2' 'http/1.1').

```
serverContext addExtension: alpn.
```

2. Configure the Listener to use Protocols.HTTPv20. E.g.:

```
listener protocolVersions: (Array with: Protocol.HTTPv20 new).
```

If a TLS connection is negotiated with the 'h2' extension, the Listener upgrades the connection from HTTP/1.1 version to HTTP/2.

Instances of Protocols.HTTPv20 include an instance of HTTP2Settings with the following options:

#outputWindowSize

Defines the buffer size for sending responses on a server. The default is defined by SettingsFrame SETTINGS_INITIAL_WINDOW_SIZE which is 65535.

#ackTimeout

Defines the amount of time before signalling a connection error. If the sender of a SETTINGS frame does not receive an acknowledgment. If the duration is not specified (i.e., nil), the multiplexer doesn't set a timer.

#settingsFrame

A SettingsFrame which implements the HTTP2 spec SETTINGS frame. For details, see the class-side defaults message category on class Protocols.SettingsFrame.

The settings frame can specify the following options:

#initialWindowSize

Indicates the sender's initial window size (in octets) for stream-level flow control.

#headerTableSize

Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific

to the header compression format inside a header block. The initial value is 4,096 octets.

#disablePush/enablePush

This setting can be used to disable server push 0 - not supported. The initial value is 1, which indicates that server push is permitted.

#maxConcurrentStreams

Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

#maxFrameSize

This setting can have any value between 2^14 (16,384) and 2^24-1 (16,777,215) octets, inclusive. All implementations MUST be capable of receiving and minimally processing frames up to 2^14 octets in length.

#maxHeaderListSize

This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field. For any given request, a lower limit than what is advertised may be enforced. The initial value of this setting is unlimited.

For example, to confugure the settings for an HTTP/2 connection:

```
(server := Server id: 'TestH2SecureServer') addResponder: Hello new.
listener := server listenOn: 8002 for: SiouX.HttpsConnection.
h2Settings := Protocols.HTTP2Settings new.
h2Settings
 enablePush;
 inputWindowSize: Protocols.SettingsFrame SETTINGS_MAX_WINDOW_SIZE;
 maxFrameSize:  Protocols.SettingsFrame  SETTINGS_MAX_FRAME_SIZE.

listener protocolVersions: (Array with:
```

```
(Protocols.HTTPv20 new
 settings:  h2Settings;
 yourself)).
```

For guidelines on optimizing your web server, "Optimizing Application Delivery" in Ilya Grigorik's *High Performance Browser Networking* (2013) is highly recommended.

Fortifying a Listener

SiouX provides a means to fortify a Listener against high-load conditions or deliberate attacks that can exhaust server resources. These measures can be applied to each Listener object, and should be set before a server is started.

To illustrate, let's use the Hello example:

```
server := Server id: 'Examples'.
server addResponder: (responder := Hello new).
listener := server listenOn: 8000 for: HttpConnection.
```

By default, the length of an HTTP request status line is limited to 8K bytes. To change the limit, specify a number (in bytes):

```
listener requestStatusLineLimit: 50.
```

By default, the length of an HTTP request header is limited to 100K bytes. To change the limit, specify a number (in bytes):

```
listener requestHeaderLimit: 200.
```

By default, there is no limit on the length of an HTTP request body. To set a limit, specify a number (in bytes):

```
listener requestBodyLimit: 102400.
```

To set an upper limit on the number of simultaneous connections:

```
listener upperConnectionLimit: 200.
```

For secure connections, a Listener may be fortified by specifying server names that are acceptable:

```
listener acceptedServerNames: #('myhostname').
```

The specified name(s) are used to validate the TLSServerName extension from a client hello and, later, the Host header field for HTTP/1.1, or the :authority pseudo-header for HTTP/2.

For secure connections, the accepted server names are part of the TLS context settings. E.g., if a server certificate is created for *.mydomain.com, the Listener can limit the accepted server names:

```
listener acceptedServerNames: #('x.mydomain.com' 'y.mydomain.com').
```

For this configuration, the Listener only accepts TLS connections only with a Server Name extension (SNI) of #('x.mydomain.com' 'y.mydomain.com'). By default, an error is raised for a connection attempt using any other name. For details, see the method comments on TLSServerContext class >> defaultServerNameVerifierValue.

It is also possible to configure your own verification behavior, e.g.:

```
listener
 acceptedServerNames: #('myhostname')
 verifier: [[:tlsConnection :receivedHostNames :acceptedServerNames | "do custom
 verification" ]. ]
```

Once the Listener has been configured, start the server as usual:

```
server start.
```

Connection

When a Listener receives a request for a new connection, it creates a new Connection object and opens it. As long as this Connection is open, it maintains a background process that handles incoming requests on an IP socket, passing them to a Server for execution. Similarly, it passes responses to an output socket. A Connection is closed either following the dictates of the protocol, or when the Server is shut down.

In fact, class SiouX.Connection, like class SiouX.Listener, is abstract. Its subclasses — HttpConnection and HttpsConnection — are used to provide support for a specific type of connection (i.e., regular or secure). The type of connection a Server supports is specified when setting up a Listener, e.g.:

```
server := Server id: 'Examples'.
httpListener := server listenOn: 8000 for: SiouX.HttpConnection.
```

In general, your application does not need to message the Connection object, as its life-cycle is managed by the SiouX framework. The options to adjust the behavior of a server's connection are provided via the Listener object. In this way, each new connection on a particular server will have consistent behavior.

For example, to specify the priority of a connection relative to other processes in the server image:

```
httpListener connectionPriority: Processor activeProcess priority + 1.
```

The SiouX framework provides several different implementations of the protocol (e.g., Xtreams and Net), and different versions of each protocol (HTTP/1.1, HTTP/2). The implementation is determined by loading the appropriate component (SiouX-Http for the Xtreams implementation, and SiouX-Net-Http for the older Net implementation), while the supported version(s) of the protocol are specified via protocolVersions:, as illustrated above.

Responder

Responders provide the criteria by which a server object accepts web requests, and the basic logic for dispatching them to the application code that handles the requests.

SiouX supports two distinct implementations (subclasses) of class Responder:

HttpResponder

Extends class Responder with implementation bits that rely on the HTTP implementation from the SiouX-Http package.

Subclasses receive SiouX.HttpRequest objects and have to create SiouX.HttpResponse objects when executing them.

NetHttpResponder

Extends class Responder with implementation bits that rely on the Net HTTP framework. Subclasses receive Net.HttpRequest objects and have to create Net.HttpResponse objects when executing them.

The choice between these implementations depends on whether you are trying to use existing application code from a previous version of VisualWorks. If so, you may need to use NetHttpResponder and or one of its subclasses.

For example, if you are doing web development with AppeX, your application needs to use subclasses from AppeX.Application, which is derived from SiouX.HttpResponder. If, by contrast, you wish to run an existing Seaside or WebServices application with SiouX, use NetHttpResponder or one of its subclasses.

Configuring Responders

There are two general approaches to configure a Responder: (1) using method pragmas; (2) using the Configuration Tool, and saving XML configuration files.

Configuring Responders with Pragmas

A single server object may be configured with a number of distinct responder objects. Method pragmas may be used in extension methods to class Responder, to configure its behavior. The two-argument pragma #server:path: is used to define the ID of the server that uses the responder, and the URL path that it will accept. This is the minimal configuration for a responder, but additional pragmas may be added to specify a more detailed configuration.

Before responders created with pragmas are added to a server, they are sorted based on their path. The sorting is performed Responder class >> isPragma: pragmaA configuredBefore: pragmaB. By default, the algorithm sorts the deepest path first. A Responder with the default path: '/' is the last one.

The following pragmas are optional. They only apply to methods that already use the #server:path: pragma:

#requestFilter:

Specifies an input request filter. This is expected to be the name of a class, and a subclass of SiouX.RequestFilter. It is possible to include more than one #requestFilter: pragma.

#requestFilter:configuration:

Specifies an input request filter with a specific configuration. The configuration parameter identifies a request filter method with the pragma that should be executed to configure the filter.

To illustrate the use of these optional pragmas, consider the LocalHello responder from the SiouX-Examples package. Its configuration is declared using the following two methods:

```
LocalHello class>>serverConfiguration
 <server: 'Examples' path: '/local-hello'>
 <requestFilter: #'SiouX.LocaleFilter' configuration: 'LocalHello configuration'>

LocaleFilter >> localHelloConfiguration
 <configuration: 'LocalHello configuration'>
 self allow: '*'.
```

Here, the LocalHello responder is configured with the request filter LocaleFilter to enable setting the locale during the processing of each request. LocaleFilter extracts the locale information from the headers of an HttpRequest, based on a customizable set of rules. I.e., when a language tag is extracted from a request with the language code 'en', the filter is accepted and a corresponding locale will be set.

Example

To start the example server that is configured using pragmas:

```
(server := SiouX.Server id: 'Examples') start.
```

This server collects responders with corresponding pragmas. To see the responders defined by the example, inspect:

```
server responders.
```

To try the example server and see its multilingual responses, open a Web Browser on this URL:

```
http://localhost:8888/local-hello
```

When you are finished with the example, release the server:

```
server release.
```

Creating Responders using the Configuration Tool

Create, modify and delete responders in the Configuration Tool by selecting a Server and choosing an option from the **Responders** menu.

A responder path is editable and can be changed even when the server is running.

The following options are available on the **Responders** menu:

Create…

Add an instance of an existing Responder subclass to the Server.

Add…

Add a preconfigured Responder to a Server. This option is disabled if the Server has already included all predefined responders.

Edit

Currently active for instances of NetHttpResponder only. A dialog prompts to set the attachment directory and an option to chunk HTTP messages.

Request Filters…

Open a dialog to add or remove filters from the selected responder. For details, see: Configuring a RequestFilter.

Change…

Open a dialog to set the options for a NetHttpResponder. For details, see: Configuring a NetHttpResponder.

Delete

Delete the selected responder.

Open Browser

Open the default Web Browser for the responder path. The server has to be running.

Open Class Browser

Open the default Web browser for the selected responder. If none selected, the browser opens on the first responder.

The Configuration Tool may also be used to set all parameters of Listeners. For details, see: Configuring a Listener.

Configuring a NetHttpResponder

The dialog to configure a NetHttpResponder includes settings for the following options: **Save attachments as files; Specify attachments directory;** and **Chunk responses larger than the specified size.**

Responder Implementation

When a Server receives a #dispatchRequest: message from a Connection, it queries each of its responders to determine whether they will accept the request. If one of the SiouX.Responder objects accepts the request, it is used to dispatch the request. The Responder takes over the connection (for that request only), reads the request in, applies request filters, executes the request and sends a response back.

Typically, the contents of the incoming request's status line (e.g., method, URL path, protocol version) are used to dispatch it. By default, a request is accepted if the request URL path matches that of the responder, but you may create custom Responder subclasses that override the #acceptRequest: method to define their own acceptance criteria.

Defining a Custom Responder

Class Responder is itself abstract, but it defines the basic structure of request processing.

Your application can implement specialized responder behavior by subclassing one of two specific implementations of class Responder in the SiouX framework: HttpResponder or NetHttpResponder.

By default, a request is accepted by a responder if its path begins with the same string as that of the responder path. If your application requires more complicated criteria, or if you wish to use special error handling for filter errors or custom error responses, you might consider a custom subclass. Typically, your subclass can implement its own #acceptRequest: method to define special acceptance criteria. Similarly, by implementing #applyRequestFilters:, your subclass can provide different criteria for applying filters.

Since a single Responder may be handling concurrent requests from multiple connections, any request-specific state *must* be treated in a thread-safe manner. Specifically, the instance variables in responder classes *are not* thread-safe unless specific precautions are taken.

To define your own Responder, create a subclass from either HttpResponder or NetHttpResponder, define the application responder path and implement at least the #executeRequestFor: method to process client requests. Subclasses are expected to implement the following additional methods:

createRequestFrom: aRequestLine

> This step follows acceptance of a request and its purpose is to build and return a request object for execution

executeRequestFor: aRequestContext

> A responder has already accepted the request and should process it here. The parameter is an instance of RequestContext that holds an incoming request and connection. After processing the request the Response is returned in aRequestContext parameter.

sendResponse: aRequestContext

After a request is executed and a response is created, this method is responsible for transmission of the response through the specified connection. Note that transmission may involve further content generation, as parts of the response content may be written directly to the outgoing socket to ensure scalability. The method has to return the message body size which is used by a CommonLog if it is turned on.

newResponse

Return an instance of the appropriate Response class for the Responder.

RequestContext

Instances of RequestContext hold all objects related to a specific request: a request, connection and responder.

An instance of the RequestContext is created by a Responder, and passed as a parameter to the Responder>>#executeRequestFor: method.

RequestFilter

RequestFilter objects provide a more flexible way of configuring responders. For example, using a filter, a single responder class can be configured to work with or without a session or locale filter.

Configuring a RequestFilter

Using the Server Configuration Tool, you can add or remove request filters to specific responders. With a Responder selected, pick **Request Filters…** from the **Responders** menu to open a list of all filters associated with the responder.

Withion this dialog, you may **Add** or **Remove** filters to/from the responder. E.g., the **Add** button opens a list of existing filter classes and configurations. Select the filter and configured filter and click **OK** to add it to the responder. Use the arrow buttons to change the priority of the filter.

Using a RequestFilter for HTTP Authentication

A RequestFilter may be used to implement basic authentication over HTTP or HTTPS. For example, a RequestFilter can be implemented that protects a responder by requiring a username and password from the client. The filter checks if an Authorization header field is included in the request, and if the field contains the required authentication data. If the request doesn't include the required credentials, the filter raises an exception, to send an Authentication request to the client.

Note that using basic authentication with standard HTTP, the password is sent in clear text. basic authentication. For better security, use an encrypted HTTPS connection.

To illustrate this use of a RequestFilter, SiouX includes class BasicAuthenticationFilter in the SiouX-Examples parcel. (Load this using the Parcel Manager.)

The BasicAuthenticationFilter performs simple authentication against the username **user**, the password **SiouX**, in the realm **SiouX**. You can browse BasicAuthenticationFilter>>localHelloConfiguration. to see how this is handled in the request filter. (Naturally, a production application would handle the password differently, but this is an example.)

The following sections explain how to configure this example request filter using the Configuration Tool or using code.

Adding a BasicAuthenticationFilter with the Configuration Tool

To add a BasicAuthenticationFilter filter to the Hello example using the Configuration tool:

1. Select the Hello responder and choose **Request Filters**... from the **Responders** menu.

   This opens a separate editor window containing list of request filters for the selected Responder. You can **Add**, **Remove**, or change the priority of any filters from this window.
2. Click on **Add**... choose **BasicAuthenticationFilter with configuration: 'Test Basic authentication'** in the dialog, and then click **OK**.
3. When finished, close the **Request Filters** editor.

   Your changes are automatically accepted.

With the request filter added, the browser will authenticate your access the to Hello example.

Adding a BasicAuthenticationFilter programmatically

First, create and configure a server, add a responder, and then a request filter. This can be done easily in a Workspace, e.g.:

```
(server := Server id: 'TestAuthentication')
 listenOn: 8005 for: HttpConnection;
 addResponder: (responder := Hello new).
responder
 path: '/hello';
 addRequestFilter:
  (RequestFilter fromConfiguration: 'Test Basic authentication').
```

Start the server:

```
server start.
```

Note that the following code now raises a Net.HttpUnauthorizedError:

```
HttpClient get: 'http://localhost:8005/hello'.
```

To satisfy the authentication filter, add a user name and password to the client, e.g.:

```
aClient := HttpClient new.
aClient username: 'user' password: 'SiouX'.
```

Trying again, the client should receive a successful response:

```
aClient get: 'http://localhost:8005/hello'.
```

Finally, release the server:

```
server release.
```

Implementation

Each RequestFilter object receives a request via the #applyTo: method. This method either processes or denies the request. If the request

is processed, by default there is no return value to the Responder, since ordinarily the Responder doesn't validate the filter output. (This behavior may be changed by overriding #applyRequestFilters: in a custom Responder subclass.) A Request will be processed only if none of the request filters raises an exception.

If a filter wants to stop processing a request, it should create a response and raise ResponseReady exception. This exception is raised by using the HttpResponse>>signalReady method, e.g.:

```
BasicAuthenticationFilter>>replyUnauthorized: aRequestContext
"Check if the Authorization header field is present."
"If there is no Authorization field, return a 401 (Unauthorized) response."

aRequestContext response
 code: 401;
 authenticate: (Authenticate basicRealm: realm);
 signalReady.
```

There are several examples of a request filter implementation, in the SiouX-Examples and SiouX-WebSocket packages:

PAMAuthFilter

Authenticates a user using the LinuXPAM libraries.

ETagRequestFilter

This is part of the AppeX-Core package. ETagRequestFilter may be added to an application responder, to enable client-side resource caching through the use of the ETag response header as described in RFC 7232.

NoCacheRequestFilter

This is part of the AppeX-Core package. This request filter ensures that the specified services set their HTTP response headers to indiacte no caching.

Defining a Custom RequestFilter

To create a new RequestFilter class for use in your application, create a subclass of SiouX.RequestFilter and implement the method #applyTo:. The expected paramater is a RequestContext. For a working example of this method, browse LocaleFilter>>applyTo:.

Optionally, you can change the ordering of the filters by overriding the implementation of `#addToResponder:` in your subclass. By default, this method adds each `RequestFilter` to the end of the collection, but you may wish to add new ones at the beginning.

To create a new filter configuration, you must create an annotated instance-side method that includes a `#configuration:` pragma, e.g.:

```
<configuration: 'Filter Configuration name'>
```

When a new instance of the filter class is created, this method marked with the `#configuration:` pragma is evaluated to set the filter options.

Chapter

3

Requests and Responses

Topics

SiouX can interoperate with two distinct frameworks for handling web requests and responses: (1) Net, which provides backward compatibility for existing VisualWorks applications, and (2) Xtreams, which provides a new, high-performance alternative. Going forward, VisualWorks development will be based upon the Xtreams framework.

To distinguish between the Net and Xtreams implementations the documentation will at times reference their respective name spaces, `SiouX.*` and `Net.*`, i.e., `SiouX.HttpResponse` versus `Net.HttpResponse`.

This chapter describes the implementations of the `HttpRequest` and `HttpResponse` classes. For details on the `Net.*` classes, refer to the *Internet Client Developer's Guide*.

HttpRequest

Class SiouX.HttpRequest employs an on-demand approach to read HTTP messages. In this way, it operates more efficiently than class Net.HttpRequest.

When an incoming request is received, the following header fields are parsed immediately: Transfer-encoding, Content-length, Connection, Content-type, Content-disposition, Content-encoding. These fields are used to remove transfer encoding, and to prepare for reading and decoding the message body.

The remaining fields are parsed on demand. The HttpRequest instance variable #headers contains a collection of HttpRequestHeaderField objects. The HttpRequestHeaderFieldinstance holds the header field name and the header value as a ReadStream.

For example, the HttpRequest>>cookies method finds all HttpRequestHeaderFieldinstance objects with the string 'cookie' in their names, and uses class Cookie to parse the stream of header values.

Implementation

Class SiouX.HttpRequest defines the following general-purpose API methods:

#contentType

Returns the Content-Type header.

#charset

Returns the Content-Type charset or nil if the request doesn't specify it.

#cookies

Returns a Collection of Cookie instances.

#hasBody

Returns a Boolean indicating that the request has a message body.

#isConnectionClose

Returns a Boolean that depends on the value of the Connection field in the HTTP header (i.e., returns true if the header value is close).

#isMultipart

Returns a Boolean indicating whether the message is simple or multipart.

Reading the Content of an HTTP Request

Because the contents of a HTTP request may be arbitrarily large, the body of any message is always provided as a stream. The contents of the stream are fully decoded according to the Content-type and Content-encoding fields. That means if the content type is textual, the stream will yield characters, otherwise it will yield bytes.

A fundamental property of the SiouX.HttpRequest implementation is that the body stream is set up directly on top of the socket, and consequently is non-positionable and can be read only once. If the content is needed more than once, it must be copied somewhere as it is being read.

For multipart messages, only one part is "active" at any given time. This means you can only process the body of the currently-active part, and once you move to the next part, you can't go back to the previous one. However, you can hold onto the parts themselves and access their headers, but not their bodies, at least of those which are already gone.

To illustrate some more features of class HttpRequest, consider the following example:

```
"Create a stream on an example request"
aString :=
'Host: localhost
Content-type: multipart/form-data;boundary="=_vw0.15035853588505d_="
Content-length: 244

--=_vw0.15035853588505d_=
Content-type: text/plain;charset=utf-8
Content-disposition: form-data;name=text;filename=text.txt
```

```
some text
--=_vw0.15035853588505d_=
Content-disposition: form-data;name=foobar--=_vw0.15035853588505d_=--'.
bytes := (ByteArray new writing encoding: #'ISO8859_1') setLineEndCRLF;
 write: aString;
 close;
 terminal.
request := SiouX.HttpRequest path: '/test' method: 'POST' version: '1.1'.
request from: bytes reading.
```

Here, the main part of the `request` has three headers, i.e.:

```
request headers size = 3
```

To read the first part, evaluate:

```
part := request body get.
```

To read the body of the first part:

```
value := part body rest.
```

To read the second part:

```
part := request body get.
```

This part should have a single header, e.g.:

```
part headers size = 1
```

If the second part doesn't specify an encoding, we can use `#'iso-8859-1'` to read and decode the body like this:

```
(part body encoding: #'iso-8859-1') rest.
```

An attempt to read the next part can raise an `Incomplete` exception:

```
[request body get] raise: Incomplete.
```

Now, close the body stream and set the source to `nil`:

```
request release.
```

Next, let's consider the example of reading a `POST` message with form data. We can build a minimal request, and then see how `#getFormData` parses it.

```
aStream := (ByteArray new writing encoding: #utf_8)
      setLineEndCRLF;
      yourself.
aStream
  write: 'Host: localhost'; cr;
  write: 'Content-type: application/x-www-form-urlencoded; charset=utf-8'; cr;
  write: 'Content-length: 26'; cr; cr;
  write: 'author=Virgil&title=Aeneid';
  close.
request := SiouX.HttpRequest path: '/test' method: 'POST' version: '1.1'.
request from: aStream terminal reading.
data := request getFormData data.
```

The value of `data` is an `Array` of `Association` objects, one for each parameter in the form. E.g.:

```
#('author' -> 'Virgil' 'title' -> 'Aeneid')
```

Note, again, that `#getFormData` can be sent only once for any given request object.

Reading HTTP Header Fields

To read the values of the header fields, use `HttpRequest>>#headers:do:`.

The `#headers:do:` method evaluates the specified block for each occurrence of a header field with the specified name, and it returns a count of the number of occurrences.

For example, to read the HTTP header `'sec-websocket-version'`, your application could add the following extension method to class `SiouX.HttpRequest`:

```
wsVersion
 self headers: 'sec-websocket-version' do: [:field | ^field body rest].
 ^nil
```

Reading Cookies

Support for cookies under SiouX follows the RFC 6265 specification.

From the developer's perspective, request cookies are represented as instances of class Cookie, where a single instance holds all the cookies associated with an incoming Request object. Each individual cookie is a name-value pair of String objects, and thus an instance of class Cookie may be viewed as a dictionary of names and values.

To access these pairs, use: Cookie>>parametersAt: aString, e.g.:

```
value := Cookie from: 'SID=31d4d96e407aad42; lang=en-US' reading.
(value parametersAt: 'lang') = 'en-US'.
(value parametersAt: 'SID') = '31d4d96e407aad42'.
```

HttpResponse

Class SiouX.HttpResponse offers both greater performance and flexibility than its counterpart in the Net framework. Instances contain a header and a body, yet while the header always represents a sequence of response header fields, the body may be representing using several different forms.

Generally, web applications that are designed for an international audience send XHTML using a Content-type of text/plain; charset=UTF-8. Accordingly, this is the default Content-type of an HttpResponse.

To create a simple text message in UTF-8:

```
response := HttpResponse code: 200.
response contents: 'Nazdar sv#te, jak se máš'.
```

It is of course possible to specify the ASCII character set, though UTF-8 is recommended to avoid encoding issues:

```
response := HttpResponse code: 200.
(response contentType: 'text/html') charset: 'ascii'.
response contents: '<html><head>Hello</head></html>'.
```

The body of an HttpResponse can be an internal collection, such as a ByteArray or String. It can be a ReadStream, which is copied directly to the

socket stream when the part body is being sent to the client via the socket. It can also be a Filename, in which case the contents of the file are sent to the client in an efficient manner.

For example, to create a simple response from a file:

```
response := HttpResponse code: 200.
image := Kernel.ObjectMemory imageName asFilename.
(response contentDisposition: 'attachment') fileName: image tail.
response contents: image.
```

Finally, a response body can be a block that is evaluated with a WriteStream as its argument, and into which it is expected to write the content. The stream argument may be set up as either a character or byte stream, according to the Content-type of the response. Again, the block is evaluated when the body content is being sent to the client, not when it is assigned as the part body.

To create a simple response with a body that is generated by a Block:

```
response := HttpResponse code: 200.
response contents: [:aStream | 10 timesRepeat: [aStream put: 42]].
```

Setting Cookies

Support for cookies under SiouX follows the RFC 6265 specification.

Response cookies are represented as instances of class SetCookie, where a single instance holds all the cookies associated with a Response object. Each individual cookie is a name-value pair of String objects. To set these pairs, use: SetCookie>>parametersAt: aString put: aValue.

To set the other attributes of a response cookie, the following behavior is provided: #expires: aTimestamp; #maxAge: aNumber; #setSecure; #domain: aString; #setHttpOnly; #path: aString.

To illustrate:

```
setCookie := SetCookie name: 'lang' value: 'en-US'.
setCookie
 path: '/';
 domain: 'example.com';
```

```
expires:
 (Timestamp readFrom: 'Nov. 1 2012 12:01:01 GTM' readStream);
maxAge: 300;
setSecure;
setHttpOnly.
```

This code sets the following response header:

```
Set-Cookie: lang=en-US; Path=/; Domain=example.com; Expires=Thu, 1 Nov 2012
 12:01:01 GMT; Max-Age=300; Secure; HttpOnly
```

Sending MultiPart Responses

Class SiouX.HttpResponse provides support for a multipart response body, which is represented using an OrderedCollection. Parts are added to the response by sending it #addPart:, which takes the desired content as its argument. The return value is an instance of class HttpResponsePart, which may be further customized by adding additional header fields. An instance of HttpResponse that was initially created with simple body is automatically converted to a multipart response if it subsequently receives the message #addPart:.

The response body is sent out either with a specified Content-length or in chunked form. If the total byte length can be determined straight away without much overhead, the Content-length field is generated automatically by HttpResponse. Otherwise, the response body is chunked.

To create a multipart response:

```
response := HttpResponse code: 200.
(response contentType: 'multipart/related') boundary: '######'.
```

To add a text part:

```
part := response addPart: 'Hello'.
```

If the part body is a string, the part is created with Content-type is text/plain by default:

```
part contentType type = 'text'.
part contentType subtype = 'plain'.
```

To add a binary part:

```
part := response addPart: (ByteArray new: 10 withAll: 42).
```

If the part body is binary, the part is created with `Content-type` is set to `application/octet-stream` by default:

```
part contentType type = 'application'.
part contentType subtype = 'octet-stream'.
```

To write the response and close the stream:

```
response writeOn: out.
result := ((out close; terminal) reading encoding: #ascii) rest.
```

Chapter

4

Sessions

Topics

Since HTTP is by nature a stateless protocol, application servers generally provide support for *session objects* that may be used to track each client's status or progress through the different stages of a web application. By establishing a distinct session for each new client, a web application can give meaningful continuity to all subsequent transactions with the same client. Typically, a session object is created to track user-identification and user-specific data, and it exists until either the client concludes the session (i.e., by logging out), the session timeout expires, or the server concludes it.

SiouX provides modular, optional support for sessions in the SiouX-Sessions parcel, which contains all classes and code required to create and manage HTTP sessions.

Implementation

The core classes comprising session support are: Session, SessionCache, SessionCachingRule, and SessionFilter. The session framework also includes several announcement classes to represent information events concerning the session cache.

When support for sessions is loaded, each incoming request is processed using a SessionFilter, to extract a unique session identifier. This identifier is in turn used to look up a Session object in a session cache, and to add it to the current RequestContext. If a session is not found in the cache, the filter creates a new Session instance, adding it to the cache, as well as to the RequestContext.

Session

Instances of class Session represent unique client sessions. That is, for each client session there is a single corresponding session object held in a SessionCache.

Instances respond to the following messages:

id

Returns a String that uniquely identifies the session.

status

Returns a Symbol indicating the current state of the session: #active or #expired. Subclasses may define other status values.

creationTime

Returns a Timestamp representing the session object's creation time (in UTC).

lastUsed

Returns a Timestamp representing the last time the session was used before its expiration (in UTC).

The following messages provide access to additional, optional information:

encoding

Returns a Symbol indicating the encoding used by this session, or nil if none has been specified.

locale

Returns the Locale object assigned to this session, or nil if none has been specified.

An instance is created using one the following two API methods:

id: aValue

Return a new instance with the specified ID, where aValue represents the ID as a String.

newWithID

Return a new instance with a unique id generated by SessionIDGenerator, which uses FastRandom.

To change the status of a Session object, use #activate. This sets its #status to #active if the session is not expired or raises an ExpiredSessionError. To expire a session, use #expire. By sending #touch, the lastUsed variable is set to Timestamp now, unless the session has already expired.

To check the session's current status, use #isActive and #isExpired.

Examples:

```
cache := SessionCache new.

session := Session newWithID.

cache rememberSession: session.
```

SessionCache

Instances of class SessionCache provide a reentrant mechanism to cache, lookup and release Session objects. Each of these actions results in a notification being sent to a collection of rules (instances of class SessionCachingRule) that govern the specific behavior of the cache. In addition, each time a session object is cached or released, a SessionCached or SessionReleased announcement is processed.

Session timeouts are implemented on a per-cache basis. That is, every session object stored in the same cache has the same timeout interval. Creating an instance of class SessionCache starts a process which uses the interval stored in its #pruningFrequency variable to periodically perform pruning actions.

A SessionCache instance should be configured with a set of SessionCachingRule objects, using the #addRule: method. Each time the cache is pruned, these rules receive a notification about the individual session objects being pruned. The caching rules in turn determine which actions should be performed on the session objects (e.g., what happens when a session expires), based on specific conditions implemented by the rules. When a session is being pruned, the SessionCache announces it with an instance of class SessionPruning.

Class SessionCache defines the following class-side API methods:

pruningFrequency: aDuration

Sets the default pruning frequency for newly created instances of SessionCache. The default value is 1 minute.

pruningFrequency

Returns the default pruning frequency.

Class SessionCache instance-side methods:

addRule: a SessionCachingRule

Add a subclass of SessionCachingRule. A rule determines which actions should be performed on the cached session objects.

findSession: sessionId

Find the session matching sessionId.

pruneEvery: aDuration

Change the pruning frequency.

releaseSession: aSession

Release aSession from the cache and announce SessionReleased.

rememberSession: aSession

Add aSession to session cache and announce SessionCached.

removeRuleOf: a SessionCachingRuleClass

Replace an existing rule whose class is SessionCachingRuleClass with nil. This effectively removes it from the rules.

sessionTTL: aDuration

Replace the existing SessionInactivityRule with a new one, which specifies the inactivity period as a Duration.

startPruning

Start the process that prunes sessions.

stopPrunning

Stop the process that prunes sessions.

Examples:

```
cache := SessionCache new
 pruneEvery: 30 seconds;
 addRule: (SessionInactivityRule withDuration: 10 minutes);
 yourself.
```

A previously-configured caching rule can be removed or replaced with a different rule. When a rule is being added to a cache, it will automatically replace an existing rule of the same class. Thus, one only needs to add/remove rules:

```
cache removeRuleOf: SessionInactivityRule.
cache addRule: (SessionInactivityRule withDuration: 1 minutes).
```

To add, lookup, and release sessions from the cache:

```
cache rememberSession: aSession.

cache findSession: 'a session id'. "Typically, a session id is a string, but it may be
 application-specific."

cache releaseSession: aSession.
```

Interested objects may receive notifications about the caching events via announcements, for example:

```
cache when: SessionCached do: [:ann | session := ann session].
cache when: SessionPruning do: [:ann | session := ann session].
cache when: SessionReleased do: [:ann | session := ann session].
```

SessionCachingRule

SessionCachingRule is an abstract class whose instances accept change notifications from a SessionCache. Instances of SessionCachingRule can collect information on these changes and alter the SessionCache, perhaps removing old, unused sessions, in accordance with a desired caching behavior. For example, the subclass SessionInactivityRule represents a caching rule to expire sessions after some period of inactivity.

Subclasses of SessionCachingRule must implement:

aboutToPrune: aSessionCache

Notification that aSessionCache is being pruned. Subclasses can override the method to add specialized behavior.

pruningSession: aSession

Notification that aSession is being pruned. Subclasses can override this method to perform actions on the session object.

accessedSession: aSession

Notification that aSession has been accessed through the cache. Generally, that means that aSession is active and in use.

addedSession: aSession

Notification that aSession has been added to a cache. By default, aSession receives the #touch message.

removedSession: aSession

Notification that aSession has been removed from a cache.

For example:

```
SessionCache new
```

```
addRule: SessionInactivityRule new;
addRule: MyServerLoadRule new;
yourself.
```

SessionFilter

SessionFilter is an abstract class for filters that extract session information from an HTTP request, and provide access to sessions cached in the filter's session cache. SessionFilter objects are added to a Responder as a kind of request filter.

A SessionFilter instance creates and holds an instance of class SessionCache. When a session id is extracted from an HTTP request, the filter tries to find a session object using #findSession:. If this fails, it creates a new instance. The default session class is Session.

Subclasses must implement the following messages:

extractSessionIdFrom: aRequestContext

Given a RequestContext, return a String session ID, or nil.

Examples that illustrate the use of class SessionFilter can be found in the SiouX-Examples parcel. For example, class SessionQueryFilter extracts the session id from query data using the query parameter sessionID as a key.

For example:

```
aResponder
 addRequestFilter: (sessionFilter := SessionQueryFilter new);
 addRequestFilter: (LocaleFilter fromConfiguration: 'configureLanguagesForTesting').
```

Chapter

5

Serving Files

Topics

Class `FileResponder` provides a basic service for uploading and downloading files from a SiouX server, allowing you to manage files in a configured root directory.

Configuring a FileResponder

For the purpose of illustration, we can create a FileResponder using Workspace code.

To create and configure a server with a FileResponder, first load the SiouX-Http parcel, if you have not already done so. Then, evaluate the following code in a Workspace:

```
server := Server id: 'FileResponderTest'.
server addResponder: FileResponder new.
listener := server listenOn: 8888 for: HttpConnection.
server start.
```

The default URL path of this responder is /files. The default root directory for upload/download is the `/sioux/files` subdirectory of the VisualWorks `/image` directory. If necessary, this subdirectory will be created by the FileResponder. To change the default directory use:

```
FileResponder class>>rootDirectory: aLogicalFilenameOrString
```

The responder supports GET, POST, PUT and DELETE methods.

Use GET to download an existing file and POST to upload and create a new one.

GET fetches the contents of the file identified by the request URL or, if it is a directory, a textual list of the directory contents.

POST allows uploading the contents of a new file identified by the request URL. If a file or directory is created, the server returns a 201 response code. If no such file name exists, the server returns a 400 response code. Any intervening directories are created as needed (up to a configured maximum depth: 20). The default path depth can be changed as follows:

```
FileResponder class>>pathDepth: aNumber
```

POST doesn't update existing files; instead, use the PUT command for that. If a file already exists, the server returns an error for POST.

Use DELETE to remove an existing file or directory identified by the request URL.

The server returns a 400 error response if:

- the file is already exists (POST only)
- the request path contains empty or ".." path tokens
- the request exceeds the supported file path depth of the host OS

Examples

The following Workspace code illustrates a brief session of using a FileResponder with the default setup at the URL localhost:8888/files.

To GET the contents of the default directory:

```
'http://localhost:8888/files' asURI get contents.
```

Returns: nil.

```
'http://localhost:8888/files/test.txt' asURI post: 'hello'.
'http://localhost:8888/files/test.txt' asURI get contents.
```

Returns: 'hello'.

To POST a file:

```
'http://localhost:8888/files/test2.txt' asURI post: 'hello2'
'http://localhost:8888/files' asURI get contents.
```

Returns a list of file names:

```
test2.txt
test.txt
```

To DELETE a file:

```
'http://localhost:8888/files/test2.txt' asURI delete.
'http://localhost:8888/files' asURI get contents.
```

Returns: 'test.txt'.

To capture and display an exception on POST:

```
['http://localhost:8888/files/test.txt' asURI post: 'hello']
  on: HttpBadRequest
```

```
    do: [:ex | ex parameter contents].
```

Returns: `'The file already exists: sioux/files/test.txt'`.

To `POST` the result of a computation:

```
'http://localhost:8888/files/factorial.txt' asURI
    post: [:out | 10 to: 20 do: [:i | out print: i factorial; cr]].

'http://localhost:8888/files/factorial.txt' asURI get contents.
```

Chapter

6

WebSockets

Topics

WebSockets provide a means to establish a bi-directional socket connection between the client and server. This allows for low-latency communication between client and server. WebSockets are intended to supersede the use of Comet or long polling techniques. They can augment Ajax when a persistent connection is required for data to be sent from server to client.

SiouX provides a complete implementation of the WebSocket protocol. To enable this support in SiouX, simply use the Parcel manager to load the package `SiouX-WebSocket`. The standard VisualWorks distribution also includes two WebSocket examples in the `AppeX-Examples` package: class `WebSocketChat` and `WebSocketApp`.

Implementation

The WebSocket protocol defines a full-duplex TCP connection between client and server. The SiouX implementation is based on RFC 6455.

The actual data is transmitted as a series of *frames*, where each frame contains an opcode, payload length, and the actual payload data. Frames may contain data or control information about the state of the WebSocket connection. The opcode is a pre-defined number that determines the interpretation of the frame data (e.g., text vs. binary).

Once a connection has been established, data frames may be sent between client and server at any time, until one side sends a `CLOSE` frame. For security reasons, the client must mask all frames that are sent to the server. The server, however, does not mask the frames sent to the client. Frames may also be fragmented, but the SiouX WebSocket implementation resolves the masking and assembles distinct frames into a contiguous stream.

Support for the WebSocket protocol in SiouX is provided by three main classes:

WebSocket

Validates that the current HTTP request can be upgraded to the WebSocket protocol.

WebSocketConnection

Implements the WebSocket connection protocol: handshake, reading, and sending WebSocket messages.

WebSocketMessage

Represents a WebSocket message, including a data stream, information concerning its type, and a flag to indicate that the message is the final frame of data.

WebSocket

A `WebSocket` validates that the current HTTP request can be upgraded to the WebSocket protocol.

An HTTP connection may be upgraded to a WebSocket connection only if:

- The HTTP request is version 1.1.
- The request method is GET.
- The Connection header field in the request includes an Upgrade token.
- The request includes an Upgrade header field, and its value is websocket.

If the server accepts the request to upgrade, it replies with an opening handshake. If the handshake is successful the HTTP connection in the RequestContext is replaced by a WebSocketConnection that is ready to send and receive WebSocket messages.

Class WebSocket can be configured with a few options:

payloadLimit

Default payloadLimit in bytes for a single frame. Default is 33554432 bytes (32 MB). The WebSocketConnection checks the payload length against a specified #payloadLimit value. If it exceeds this limit, a DataFramingClose error is raised and WebSocketConnection is closed.

subprotocols

A collection of supported subprotocols (strings) can be specified for the WebSocketConnection. By default, the collection is empty and there is no subprotocol validation. The subprotocol values are validated during the client handshake. If the subprotocols are defined for the WebSocketConnection and the client request doesn't include any of them, the handshake fails and the connection is closed.

validateHostOriginBlock

The default validateHostOriginBlock (a BlockClosure) is a three-argument block. It expects a Responder, the Host, and Origin fields from a request. The default block returns true and doesn't validate the origin. Servers that are not intended to process input from any web page but only for certain sites should validate the Origin field. If the origin is unacceptable to the

server, the block should return false, the filter stops processing the request, and returns a HTTP 403 Forbidden reply.

WebSocketConnection

Class WebSocketConnection provides an API for sending messages to the client, and for managing open connections.

Data and control messages are handled by different methods. For example, when a WebSocketConnection receives a data message, it sends the following message to a Responder:

```
receivedWSDataOn: aStream connection: wsConnection message: wsMessage
```

In the case of AppeX, the Responder is essentially the application object. In this fashion, the server application is notified of incoming websocket messages.

The parameters to #receivedWSDataOn:connection:message: may be understood as follows. First, aStream represents a Xtreams.StitchReadStream, allowing the use of fragmented data. Data can be read from aStream until the last frame raises an Incomplete error. The second parameter, aWSConnection, is an instance of WebSocketConnection. As soon as data are read from aStream, this connection object can be used to send messages back to the client.

Finally, aMessage is an instance of WebSocketMessage, which carries the parsed web socket message header. The most important information in the message parameter is its #opcode, which provides the data type for aStream. Here, aMessage>>isText indicates that the incoming stream is textual.

Note that when using Xtreams, it is possible to pass the incoming data stream object directly to another stream. For example:

```
writeStream := String new writing.
writeStream write: aStream.
```

Here, the data from the StitchReadStream is read and then written to writeStream. The Incomplete exception is handled inside #write:, such that the developer doesn't need to provide any additional error-handling code.

Sending Messages

To send a WebSocket message to a client, use the following methods in class `WebSocketConnection`:

sendTextStream: inputStream

Sends the inputStream contents as a text message (i.e., `opcode = 1`). If the input stream size exceeds the payload limit, the steam contents is sent as a fragmented message. The stream contents are sent using UTF-8 encoding.

sendBinaryStream: inputStream

Sends the input stream contents as a binary message (i.e., `opcode = 2`). If the input stream size exceeds the payload limit the steam contents are as a fragmented message.

sendStream: inputStream opcode: anInteger statusCode: aStatusCode

Sends the input stream contents over the WebSocket connection.

opcode

Defines the interpretation of the "Payload data". The opcode values are defined as a shared variable in class `WebSocketMessage`.

For more information, see Section 5.2 of RFC 6455.

statusCode

Provides additional information for the `CLOSE` control frame.

For more information, consult Section 7.4 of RFC 6455.

sendMessage: aWebSocketMessage

Use this to send small messages. The message `#data` instance variable should provide the message contents and `#opcode` defines the message type.

Connection Management

Class `WebSocketConnection` also provides some simple protocol for managing the connection state.

Once a connection has been established, a WebSocketConnection remains open until either the client sends a control frame with the CloseType opcode, or the server closes the connection sending a CLOSE frame.

close

Send a CLOSE control frame to the client, and close the socket connection.

The underlying HttpConnection object can be obtained via the #httpConnection access method. However, while it is possible to poll the state of the HttpConnection object using #isOpen (i.e., check that the socket is not nil), a better strategy is to register interest in the ConnectionClosed announcement, and manage a closed connection at that time. This announcement is broadcast when connection is closed for any reason.

For example, the WebSocketChat example application catches ConnectionClosed announcements, and removes the closed connection from a cache, e.g.:

```
establishedWSConnection: wsConnection
 wsConnection server
  when: ConnectionClosed
  do: [:ann | self closedHttpConnection: ann connection].
```

Error Handling

All WebSocket errors are handled according to the protocol specification and should not require intervention by a developer. For example, in the event of a handshake error, the HTTP connection is simply closed, and the details of the failure are recorded by the SiouX logging service.

Handling Control Frames

The WebSocket protocol distinguishes between data and control frames, where the latter are used to query or manipulate the state of the WebSocket connection itself.

When a WebSocketConnection receives a control frame, it sends the following message to a Responder:

```
#receivedWSControlFrame: aMessage connection: wsConnection
```

Your application code is responsible for interpreting control frames. For example, your application would typically check for a CLOSE control frame and, if found, remove the WebSocketConnection object from the collection of active connections.

For a simple example of how this is done, browse the implementation of WebSocketChat class>>receivedWSControlFrame:connection: (since this example does not use web sessions, the logic is implemented as class-side behavior.)

The CLOSE frame may contain a body (the "Application data" portion of the payload data) that indicates a reason for closing, such as an endpoint shutting down, an endpoint having received a frame too large, or an endpoint having received a frame that does not conform to the format expected by the endpoint. Use aMessage>>data to see the CLOSE frame body.

Extensions to Class Responder

The SiouX-WebSocket package extends class Responder with three methods for reading and sending WebSocket messages.

Developers should implement these methods to read messages from a WebSocketConnection and then process them:

establishedWSConnection: wsConnection

> Sent when the web socket connection completes its handshake and is ready to accept messages.

receivedWSControlFrame: aMessage connection: wsConnection

> Sent when the server receives a WebSocket control frame from the client. aMessage represents the control frame header and data.

receivedWSDataOn: aStream connection: wsConnection message: aMessage

Sent when the server receives a data frame from the client. Application data can be read from aStream. aMessage defines the data type: text or binary. aStream is a StitchReadStream.

WebSocketMessage

An instance of WebSocketMessage can represent either a control or data frame.

Control frames are handled by #receivedWSControlFrame: aWSMessage connection: aWSConnection. In this case, the WebSocketMessage object includes information about the control frame headers, and possible #data for a reason that explains the CLOSE control frame. The #data instance variable of the message object includes the control frame payload as a String.

Data frames are handled by #receivedWSDataOn: aStream connection: aWSConnection message: aWSMessage. In this case, the WebSocketMessage object may be used to define a write stream to read data from aStream. A WebSocket data message is either binary or text. To query the type of data message, use #isText or #isBinary.

For example, in the demonstration class WebSocketApp, the method #receivedWSDataOn:connection:message: first checks if the message is textual (i.e., #isText = true) and, if so, it creates a text stream to read data from the WebSocketConnection. I.e.:

```
aMessage isText ifTrue: [data:= String new writing].
```

When the application receives a control frame (this happens in Responder>>receivedWSControlFrame:connection:), the WebSocketMessage object contains all the control frame information. The #data instance variable of the message object includes the control frame payload as a String.

To understand the semantics of the various fields of a WebSocketMessage, consult the discussion of the WebSocket wire format for the data transfer in Section 5.2 of RFC 6455.

Example: WebSocketChat

This example uses WebSockets to implement a simple chat room.

1. After loading AppeX-Examples, open the Configuration tool and start the AppeXExamples server.
2. The server responders must include a WebSocketChat responder. If this responder is missing, use the **Add**... menu to add it as a preconfigured responder.
3. Next, open a web browser at the following address:

   ```
   http://localhost:8888/appex/websocket-chat
   ```

   Enter a user name at the prompt, and click **Accept**.
4. Open a second web browser on the same URL, and enter a different user name.

   At this point, the two users can post messages in the **Enter Post** window, and click **Send** to broadcast them to all users.

The WebSocketChat server does the following:

- Maintain a list of all connections and users
- When a user posts a new message, use the web socket connection to broadcast the message to all users
- If a user logs out, close the user connection and send a close notification to the remaining users

WebSockets define the notion of a *subprotocol*, which allows the client and server to agree on the specific content and structure of the websocket stream. Accordingly, the WebSocketChat example server declares its subprotocol as 'chat'.

For details on subprotocols, refer to Section 11.3.4 of RFC 6455.

Working with WebSockets

The following example is based upon class WebSocketChat in the AppeX-Examples package.

Creating a Server for a WebSocket Connection

To create a server that can negotiate and establish a webscoket connection, it is necessary to use a `WebSocket`, and specify its protocol version.

To illustrate, first create a test server and responder:

```
server := Server id: 'WebSocketTest'.
server addResponder: (responder := AppeX.WebSocketChat new).
```

At this point, you have two options.

The first option is to configure a `Listener`. Using this option, all connections will be upgraded to a WebSocket.

For this, create and configure a web socket filter with the subprotocol `'chat'` and a payload limit of `10240` bytes:

```
protocol := WebSocket new.
protocol  subprotocols: #('chat').
protocol  payloadLimit: 10240.
responder path: '/websocket-app/'.
listener := server listenOn: 8002 for: SiouX.HttpConnection.
listener protocolVersions: (Array with: protocol).
```

The second option involves configuring a `Responder`. A `Responder` can be configured to accept a `WebSocket` whose protocol matches a sepcified version.

In the WebSocket Chat Demo, a connection is upgraded to a `WebSocket` if:

1. There is a responder that includes the WebSocket protocol.
2. If the WebSocket protocol is configured with subprotocol `'chat'`.

You can specify the WebSocket protocol using the `Responder`. i.e., the web socket responder should implement the following method:

```
WebSocketChat>>protocolVersions
 ^Array with:
  (WebSocket new
   subprotocols: #('chat');
   payloadLimit: 10240;
```

```
 yourself)
```

Sending and Receiving WebSocket Messages

If a server-client handshake is successful, the WebSocketConnection is ready to send and receive messages. A Responder receives the #establishedWSConnection: message right after the web socket connection finishes its handshake, and is then ready to accept messages. A Responder can begin sending messages in this method.

For example:

```
WebSocketApp class>>establishedWSConnection: aConnection
 aConnection sendMessage:
  (WebSocketMessage text: 'Server connection is ready')
```

The argument for #establishedWSConnection: is an instance of WebSocketConnection. In this example, the server simply sends a notification to the client that the server is ready to accept messages.

(In the actual WebSocketChat example, this method overrides the default implementation in SiouX.Responder which itself does nothing. This method is also used to maintain a cache of web socket connections.)

As soon as a connection is established the server can use it to send messages any time. The WebSocketChat example receives user messages via #receivedWSDataOn:connection:message:, and uses its cache of connections to pass the messages to all users.

To illustrate:

```
receivedWSDataOn: aStream connection: wsConnection message: aMessage
 | message type stream data |
 "Use a text stream because we are only expecting text data."
 data := wsMessage isText
    ifTrue: [String new writing]
    ifFalse: [^self error: 'Expected text messages only!'].
 "Read the data from the web socket stream."
 data write: aStream.
 "Client message"
 data := data contents.
```

```
"Parsing JSON data"
stream := data reading marshaling: JSON.
message := stream get.
type := message at: #type.
type = 'NewUser'
    ifTrue: [self users
        at: (message at: #user)
        ifAbsentPut: [wsConnection].
       self broadcastUserList.
       ^self broadcastMessage: data].
type = 'NewPost' ifTrue: [^self broadcastMessage: data].
```

Chapter

7

Servlets

Topics

SiouX includes compatibility support for Smalltalk servlet applications. Using the SiouX-Servlet package, you can easily port existing servlet application implemented using the now-legacy VisualWorks Web Toolkit.

This chapter briefly describes the SiouX servlet implementation, and provides a step-by-step example of porting a Web Toolkit servlet application to the SiouX framework. For more detailed documentation on the VisualWorks servlet implementation, see the *Legacy Web Application Developer's Guide*.

Implementation

The SiouX servlet implementation follows version 2.2 of the Java Servlet API. All compatibility code is contained in the SiouX-Servlet package (for details, see Loading SiouX).

Servlet support is provided by the following classes in the `Sioux.*` name space:

HttpServlet

An abstract superclass that provides the basic service framework for initializing servlet instances and processing requests.

SingleThreadModelServlet

An abstract superclass for servlets which implement the single-thread model such that each servlet instance serves only a single request.

ServletResponder

Creates an `HttpServletRequest`, finds the appropriate `WebSite` object, and dispatches requests to a `ServletHandler` for execution.

HttpServletRequest

Holds a `Net.HttpRequest` object, as well as query or form data.

HttpServletResponse

Holds a `Net.HttpResponse` object, and provides an API to add HTTP headers and set the message body.

ServletContext

Unlike instance variables in a servlet object, a `ServletContext` provides a way to share data used by all servlets in a single application.

ServletSession

Generally, servlets are stateless, but the VisualWorks implementation provides support for sessions.

ServletConfig

A ServletConfig holds the context and site (application) information.

Compatibility

While the HttpServletResponse API is backward compatible with that of class VisualWave.Response, there are some minor differences:

- The #buffered option has been deprecated. By default, using #write: always saves the response body contents in a buffered stream. For details on writing responses directly to a socket stream, see "Streaming on an HTTP connection" in the *Internet Client Developer's Guide*.
- Generally, your application should use servlet API methods to manipulate response headers, but in the event that you need to access them directly, your code should use HttpServletResponse>>httpResponse.

  For example, setting #cacheControl: is now implemented as:

```
HttpServletResponse>>cacheControl: aString
httpResponse cacheControl: aString
```

Porting Servlet Applications

To port a servlet application that was written for the Web Toolkit, you need to change the superclasses of your servlets from VisualWave.HttpServlet or VisualWave.SingleThreadModelServlet to SiouX.HttpServlet or SiouX.SingleThreadModelServlet.

You also need to provide configuration code to set up a web site. In the Web Toolkit, this was handled using configuration files. For simplicity, SiouX handles site configuration programmatically.

Example: An Event Calendar

To illustrate a working example of a servlet application that has been ported to SiouX, we have provided a simple Event Calendar

application. This is contained in the SiouX-Servlet-Demo package. You can load this using the Parcel Manager.

This package includes a simple event calendar application which uses SiouX servlets and Glorp for persistent storage. The Calendar provides a simple web interface that displays a month calendar allowing you to view, add, and edit a collection of events.

All servlets in the Calendar application are specialized subclasses of CalendarServlet. Thus, to port the application to SiouX, we only changed the definition of a single class, e.g.:

```
Smalltalk.Calendar defineClass: #CalendarServlet
 superclass: #{SiouX.SingleThreadModelServlet}
```

For details on the use and implementation of the Calendar application, refer to the comment for class CalendarServlet.

Next, you need to create and configure a Web server. You can use the **Configure Web Server** tool from the SiouX-Tools parcel, or use a legacy confirguration file (`.ini`).For the latter, load the legacy web application server, and execute the following script in a Workspace:

```
SiouX.WebSiteConfiguration
 configureFromFile: '$(VISUALWORKS)/web/',
  WebSiteConfiguration defaultConfigFileName.
SiouX.WebSiteConfiguration defaultSite configParameters
 at: 'environment' put: 'Calendar'.
```

To make the application available, create and start a server using SiouX.ServletResponder, e.g.:

```
(SiouX.Server id: 'Calendar')
 addResponder: SiouX.ServletResponder new;
 listenOn: 9009 for: SiouX.HttpConnection;
 start.
```

To open the Calendar application, browse the following URL:

```
http://localhost:9009/servlet/CalendarServlet
```

Chapter

8

Announcements

Topics

Announcements are generated by a SiouX server object, to mark events such as the establishment and completion of web connections, receipt of web requests or transmission of responses to clients.

These announcement objects are used for logging server activity and any connection or internal server errors. Announcements may thus be useful for investigating server errors, tuning server performance or gathering server statistics.

All server announcements are derived from class `ServerAnnouncement`, and are divided into the following groups: connection, listener, and responder announcements.

ConnectionAnnouncement

ConnectionAnnouncement is an abstract class for connection, request and response announcements.

ConnectionClosed

Announces that a connection has closed.

ConnectionOpened

Announces that a connection has been opened.

ConnectionFailed

This announcement is generated when:

- The connection was unexpectedly closed on the other end.
- The connection inactivity timeout expired.
- A failure in a WebSocket handshake.
- A failure initializing streams for a secure HTTP connection.

RequestAnnouncement and ResponseAnnouncement

These are notifications about request processing and response events. They include the request and response objects.

There are several announcements that concern failures:

RequestExecutionFailed

Sent by a Responder if the request processing failed.

RequestStatusLineTooLarge

Sent when request status line size exceeds the number specified by Listener.RequestStatusLineLimit, which is 8192 bytes by default.

ResponseSendingFailed

Announces that writing a response to a socket connection has failed. The connection will be closed.

ResponderNotFound

Announces that a responder for the request was not found, and a `NOT FOUND` response (404) was sent.

ListenerAnnouncement

`ListenerAnnouncement` is an abstract class for announcements concerning connection acceptance throttling, and starting or removing a listener.

Note that the handlers for these announcements run within the individual listener processes. Accordingly, they should execute quickly, and pass any computationally-intensive tasks to other processes. For additional details, see: Working with Announcements.

ConnectionAcceptanceThrottled

Announces that a listener is starting to throttle the rate of accepting new connections (the `lowerConnectionLimit` has been reached).

ConnectionAcceptanceResumed

Announces that a listener is starting to accept new connections again after being suspended (the connection count dropped below `upperConnectionLimit`).

ConnectionAcceptanceUnthrottled

Announces that a listener is now accepting new connections at full speed again (the connection count dropped below `lowerConnectionLimit`).

ConnectionAcceptanceResumed

Announces that a listener is starting to accept new connections again after being suspended (the connection count dropped below `upperConnectionLimit`).

ListenerFailed

Generated when a listener fails to initialize a new connection.

ResponderAnnouncement

ResponderAnnouncement is the abstract class for responder announcements that concern adding or removing a Responder to a Server object.

A ServerShutdownAnnouncement notifies that the server is scheduled for shutdown in a period of time specified by #duration.

The global object SiouX.ServerErrors includes a collection of all error announcements: ResponderNotFound, ListenerFailed, RequestStatusLineTooLarge, RequestExecutionFailed, ConnectionFailed, ResponseSendingFailed.

Working with Announcements

Announcements are typically handled by the background process of their corresponding connection. For this reason, any code associated with announcement processing should be re-entrant. Moreover, whenever a connection process executes an announcement handler, the connection isn't handling requests, so this handler should generally finish as quickly as possible. Any larger workload should be off-loaded to other processes.

Examples

To illustrate, let's consider some simple examples. E.g., if a server announces a RequestExecutionFailed:

```
server announce:
 (RequestExecutionFailed
  exception: anError
  responder: aResponder
  request: aRequest
  connection: aConnection).
```

This could be handled by the following code:

```
server
 when: SiouX.ServerErrors
 do: [:ann | "Process error announcement"]
 for: myObject
```

The announcement can also be triggered by a more specific condition. E.g., to catch a failed connection attempt:

```
(Server id: 'Examples')
 when: ConnectionFailed
 do: [:ann | Transcript show: ann printString; cr]
 for: anObject
```

Chapter

9

HTTP/2

Topics

HTTP/2 is a higher-performance revision of HTTP that provides a number of enhancements. It extends the existing HTTP standards, but shares all the same core concepts, methods, status codes, and so forth. At present, most of the major web browsers support HTTP/2, though a secure (TLS) connection is required.

The SiouX framework provides a full HTTP/2 implementation based on RFC 7540. The implementation supports request and response multiplexing, compression of HTTP header fields, support for request prioritization, server push, flow control, error handling and upgrade mechanisms.

Overview

Support for a HTTP/2 server implementation is distributed in the SiouX-Http2 parcel. To configure a SiouX server for secure connections you need to provide valid certificates and a private key. (For details on how to obtain them, see: Working with Certificates and Private Keys.) Since current web browsers no longer accept self-signed certificates, the SiouX HTTP/2 implemention doesn't include an out-of-the-box demonstration server.

The `/preview` directory includes a HTTP/2 client implementation in the HTTP2 parcel. This may be used to test the HTTP/2 server. Class HTTP2Client supports both HTTP/1.1 and HTTP/2 connections, and provides an API for upgrading an HTTP/1.1 connection to HTTP/2. The package comments for HTTP2 describe how to use the client with both plain and secure HTTP connections.

HTTP/2 Example

The AppeX-Examples-HTTP2 parcel includes a simple HTTP/2 demo. The package comments explain how to set a server with a certificate and private key files, obtained from a Certificate Authority (CA).

The demo visually demonstrates the performance improvement of HTTP/2 versus HTTP/1.1 for web pages that include a large number of images.

Implementation

HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded frames, which are then mapped to messages that belong to a particular stream, and all of which are multiplexed within a single TCP connection. This is the foundation that enables all other features and performance optimizations provided by the HTTP/2 protocol. The protocol uses an efficient coding for HTTP header fields and allows prioritization of requests, letting more important requests complete more quickly, further improving performance.

The HTTP/2 handshake consist of a preface and a SETTINGS frame exchange. The server connection preface SETTINGS frame might include parameters that necessarily alter how a client is expected to communicate with the server. Upon receiving the SETTINGS frame, the client is expected to honor any parameters established.

Message Framing

The basic protocol unit in HTTP/2 is a *frame* which is the smallest unit of communication in the protocol. Each frame contains a header, which at a minimum identifies the stream to which the frame belongs, and a type, where each type of frame serves a different purpose.

The subclasses of `Protocols.HTTP2Frame` provide the implementation of all protocol frame types:

HeadersFrame

Used to open a `HTTP2Stream` and additionally carries a header block fragment.

ContinuationFrame

Indicates the continuation of a sequence of header block fragments.

DataFrame

Used to transport HTTP message bodies.

PushPromiseFrame

Signals a promise to serve the specified resource.

PingFrame

Provides a mechanism for measuring the minimal round-trip time from the sender, as well as determining whether an idle connection is still functional.

PriorityFrame

Specifies the sender-advised priority of a stream.

RstStreamFrame

Signals the termination of a stream and errors.

GoawayFrame

Initiates shutdown of a connection, or signals a serious error condition.

SettingsFrame

Used to communicate configuration parameters for the connection.

WindowUpdateFrame

Implements flow control. Receivers advertise how many octets they are prepared to receive on a stream and for the entire connection. This is a credit-based scheme.

Streams and Multiplexing

Multiplexing is a technique for passing multiple request or response messages through a single TCP connection.

Multiplexing of requests is achieved by having each HTTP request/response exchange associated with its own stream. An HTTP stream is a bidirectional flow of bytes within an established connection, which may carry one or more messages, where a message is a complete sequence of frames that map to a logical request or response message.

In SiouX, HTTP/2 streams are implemented by the following classes:

Protocols.HTTP2Stream

The superclass for server and client streams. A stream provides breaking down HTTP requests and responses into independent frames and reassembling them on the other end.

SiouX.HTTP2ServerStream

Provides a server-specific implementation of push features, stream dependency and prioritization.

Server multiplexing is implemented by the following classes:

Protocols.HTTP2Multiplexer

The superclass for server and client multiplexers. The class receives, sends and processes frames, opens and closes streams, updates settings, provides encoder and decoder

for HEADER frames, supports flow control, and provides log support.

SiouX.HTTP2ServerMultiplexer

Provides a server-specific implementation. The class maintains a dependency tree for stream prioritization.

HTTP2FlowControlWindow

Implements connection and stream flow control. Flow control is based on WINDOW_UPDATE frames. Receivers advertise how many octets they are prepared to receive on a stream and for the entire connection. This is a credit-based scheme. The initial flow control value is defined in the SETTINGS frame as `Protocols.SettingsFrame SETTINGS_INITIAL_WINDOW_SIZE`. A stream reduces the window size by sent DATA frame size and increases it when a WINDOW_UPDATE frame is received. If the stream's window size is zero or negative, the stream won't be able to send any DATA frames until it receives a WINDOW_UPDATE with a window size increment.

SiouX.HTTP2Dependency

Implements a node of the `HTTP2Stream` dependency tree. Each node holds information about a parent stream and its dependencies. PRIORITY frames allow a client to express how it would prefer to receive the responses. A stream dependency within HTTP/2 is declared by referencing the unique identifier of another stream as its parent; if omitted the stream is said to be dependent on the "root stream". A SiouX server puts responses into a send queue from a parent stream, ahead of its dependencies. When an `HTTP2Server` stream is done with sending all data and closed, it is then removed from dependency tree. The parent dependent streams become dependent on the "root stream" or independent. Sending responses is only allowed to independent streams.

Header Compression

The SiouX implementation of HTTP/2 header compression follows the specification in RFC 7541. Each HTTP stream carries a set of headers that describe the transferred resource and its properties.

To reduce overhead and improve performance, HTTP/2 compresses request and response header metadata using the HPACK compression format. This format requires that both the client and server maintain and update an indexed list of previously-seen header fields, which is then used as a reference to efficiently encode previously-transmitted values.

In the SiouX implementation, class HPACKEncoder is responsible for encoding header fields, while class HPACKDecoder decodes them. Both encoder and decoder maintain static and dynamic tables. The dynamic table consists of previously-seen header fields maintained in a first-in, first-out order.

Sensitive header fields may be protected by not merely compressing them but instead by encoding their value as literals.

To specify never-indexed fields:

```
HPACKEncoder class >> neverIndexedFields: anArray
```

where anArray includes field names you don't want to be encoded. By default, the list of never-indexed fields is empty.

Announcements

The subclasses of HTTP2Announcement are used for debugging. The **HTTP2 Debug** page of the VisualWorks Settings provides an option to send selected announcements to the Transcript. To use the options on this Settings page, you must load the SiouX-Tools parcel.

The following subclasses provide specific functions:

DebugAnnouncement

Used to log debug info. The framework is delivered without creating DebugAnnouncement. To create a DebugAnnouncement use #debug: aString from a multiplexer or stream.

HTTP2ErrorAnnouncement

Used to log error messages and their debug stacks. This annoncement is created by a multiplexer when catching frame processing errors.

HTTPRequestAnnouncement

Used to annouce an HttpRequest after the header request was decoded but before the request body is processed. The **Request digest size** setting on the **HTTP2 Debug** page can be used to set a print limit for the request header and body.

HTTPResponseAnnouncement

Used to announce an HttpResponse. The announcement is created before the response is split into frames. The **Response digest size** setting on the **HTTP2 Debug** page can be used to set a print limit for the response header and body.

ReceivedFrameAnnouncement

Used to announce a received frame.

SentFrameAnnouncement

Used to announce a sent frame.

To log binary HEADERS and the content of DATA frames, enable the **Use verbose frame print** setting.

To log the name of the process, enabled the **Print process name** setting.

Using an AnnouncementCollector

The HTTP/2 framework uses multiple processes, so the output of log announcements to the Transcript doesn't always reflect the correct order of the announcements. Instead of logging to the Transcript, the announcements can also be collected using an AnnouncementCollector and reviewed later.

To set up the announcement collector:

```
ac := AnnouncementCollector new.
ac announcements: ReceivedFrameAnnouncements, SentFrameAnnouncements.
ac start.
SiouX.HTTP2ServerMultiplexer announcementPrinter: ac.
```

After processing some HTTP/2 requests:

```
ac log inspect.
```

To stop collecting announcements:

```
ac stop.
```

To reset the announcement collector:

```
ac resetLog.
```

Using HTTP/2 Push on a Server

Your application should use the `Link` header field to communicate to the HTTP/2 layer that the response resource should be pushed. The format of the `Link` header field is specified by RFC 5988. A response can specify the URL that the responder wants to push by using the `#preloadLink:` method, e.g.:

```
HttpResponse new preloadLink: '/someCss.css'.
```

A server pushes responses referenced by the `Link` header field, sending a `PUSH_PROMISE` before sending an initial HTML document to ensure that clients do not request those resources.

An example of a responder that pushes resources:

```
Responder >> executeRequestFor: aRequestContext
 | lastToken |
 lastToken := aRequestContext request url tail.

 lastToken = 'someCss.css'  ifTrue: ["pushed resource"
   ^aRequestContext response
   contentType:  'text/plain';
   contents: 'some css';
   yourself ].

 lastToken = 'someJS.js' ifTrue: ["pushed resource"
   ^aRequestContext response
   contentType:  'text/plain';
   contents: 'some js';
   yourself ].

 lastToken = 'somePng.png' ifTrue: ["pushed resource"
   ^aRequestContext response
   contentType:  'text/plain';
```

```
  contents: 'some png';
  yourself].

"Initial response that specifies resources to push"
^aRequestContext response
 contentType:  'text/html';
 preloadLink: self path, '/', 'someCss.css';
 preloadLink: self path, '/', 'someJS.js';
 preloadLink: self path, '/', 'somePng.png';
 contents: '<HTML><BODY>some html</BODY></HTML>';
 yourself
```

For guidelines on tuning server push to improve page load times, "Rules of Thumb for HTTP/2 Push" (2016) by Tom Bergan is highly recommended.

Stream Prioritization

The HTTP/2 implementation supports stream dependencies and weights. A stream dependency within HTTP/2 is declared by referencing the unique identifier of another stream as its parent; if omitted, the stream is said to be dependent on the "root stream". A server constructs a dependency tree to ensure that the parent stream's resources are allocated ahead of its dependencies.

A server can be set to ignore dependencies and instead send responses as they are received. To ignore dependencies use:

```
SiouX.HTTP2ServerMultiplexer class>>ignoreDependency: true.
```

Note: There is a known problem with creating Event streams in Chrome. An Event stream has long-lasting connection for send/receive server events. Chrome sets dependency on an EventStream. A stream dependent on Event stream will wait until the EvenStream won't be closed. The only solution right now to set the server to ignore dependencies.

For more information about HTTP/2 stream prioritization and dependencies, see: "Stream Prioritization" in Ilya Grigorik's *High Performance Browser Networking* (2013).

Memory Management

Each HTTP2 stream allocates and releases several ByteArrays to buffer request and response data. To improve performance, a Multiplexer creates an instance of HTTP2RecyclingCenter. Class HTTP2RecyclingCenter caches ByteArrays to reduce the load on the garbage collector. The recycling center maintains a cache of ByteArrays for input, output and frame buffers.

By default, the maximum size of cached buffers is 10 and is defined by Protocols.HTTP2RecyclingCenter.CacheLimit. The default can be set using:

```
HTTP2RecyclingCenter cacheLimit: aNumber.
```

Recycling is implemented as follows:

- A multiplexer receives a data frame and asks the recycling center for a data buffer.
- The recycling center checks its cache and returns an existing buffer.
- If no buffer is available, it is created.
- After a buffer has been used, the Multiplexer returns it to the recycling center, which caches it for the next usage.

Several features are available to help you with measuring and adjusting the performance of the recycling center. For example, to set the maximum size of the cache:

```
Protocols.HTTP2RecyclingCenter.CacheLimit := 10.
```

To begin collecting statistics on buffer re-use:

```
Protocols.HTTP2RecyclingCenter default startLogging.
```

To print the log statistics to the Transcript:

```
Protocols.HTTP2RecyclingCenter default logPrint.
```

The statistics can provide information on how often new memory is allocated, versus use of the recycling cache.

To stop logging statistics:

```
Protocols.HTTP2RecyclingCenter default stopLogging.
```

Errors

Two types of errors are supported: connection and stream errors:

HTTP2ConnectionError

Implements a connection error that is any error that prevents further processing of the frame layer or corrupts any connection state. A server when encounters a connection error sends a GOAWAY frame with the stream identifier of the last stream that it successfully received from its peer. The GOAWAY frame includes an error code that indicates why the connection is terminating. After sending the GOAWAY frame for an error condition, the server closes the TCP connection.

HTTP2StreamError

Implements a stream error that is an error related to a specific stream that does not affect processing of other streams. A server when detects a stream error sends a RST_STREAM frame that contains the stream identifier of the stream where the error occurred. The RST_STREAM frame includes an error code that indicates the type of error.

Chapter

10

Logging

Topics

The SiouX-Loggers package provides support for logging Server announcements. Logs are generally written to files, but the output may be alternately directed to any textual write stream, such as the VisualWorks Transcript or standard out.

Basic Logging Facilities

SiouX provides three basic classes for logging web requests, server status, and errors:

CommonLog

Class `CommonLog` implements the so-called “Common Log Format”, also known as the “NCSA Common log format”, an industry-standard log format used by many web servers.

StatusLog

Class `StatusLog` provides information on server activity and performance. It allows a server administrator to find out how well the server is performing. This log uses a `ServerStatusCollector` to monitor a server for a specified time period, collect statistics and then print the collected data.

ErrorLog

Class `ErrorLog` logs SiouX server errors. The error log data includes the date, time, remote server, request status line and Smalltalk exception stack.

These SiouX loggers are derived from class `StreamLogger`. This is an abstract logger that may be simply set up with a log file name. In this case, the logger automatically opens and closes the log file as needed. Alternatively, it can be configured with a pre-configured textual write stream in which case the stream provider is responsible for managing (i.e. opening/closing) the stream.

A logger can be suspended or active which is controlled with the `#start` and `#stop` messages. When stopped, nothing is logged. Each logger is created in the stopped state.

Each logger derived from `StreamLogger` is configured with the default file name and directory. To change the default log directory:

```
CommonLogger class>> rootDirectory: aLogicalFilenameOrString
```

To change the default file name:

```
CommonLogger class>> fileName: aStringOrFilename
```

CommonLog

This class implements the so-called “Common Log Format”, also known as the “NCSA Common log format”, an industry-standard log format used by many web servers.

The common log file format logs one line for each completed request, where the line has following structure:

```
remotehost ident authuser [date] "request" status bytes
```

The individual fields of the line have following meaning:

remotehost

Remote hostname or IP address if DNS hostname is not available.

ident

The remote logname of the user (per RFC 931).

authuser

The username the client used to authenticate itself.

[date]

Date and time of the request, %d/%b/%Y:%H:%M:%S %z

"request"

The request line exactly as it came from the client.

status

The HTTP response status code returned to the client.

bytes

The byte length of the response (without the header).

Here's a sample line of a log, with a hyphen "-" in place of a field to indicate missing information:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200
 2326
```

For more information about this format, see:

```
http://en.wikipedia.org/wiki/Common_Log_Format
```

```
http://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format
```

```
http://httpd.apache.org/docs/2.2/logs.html#common
```

StatusLog

This class provides information on server activity and performance. It allows a server administrator to find out how well their server is performing. The log uses a `ServerStatusCollector` to monitor a server for a specified time period, collect statistics and then print the collected data.

The output provides information about the number of connections, requests, responses, pending responses, rate and bytes sent per specified period of time.

The server statistics are printed in the following format:

```
##Monitor interval: 1 minute
=====================================================
Connections    Requests    Responses    Pending    Rate  Bytes Sent (K)
=====================================================
15:51 0001     00050        00050          0000        0001     00000021
```

Monitor interval: the period of time to collect statistics.

The value for `Rate` is calculated as (responses * 1000 / monitor interval asMilliseconds), while that of `Bytes Sent` is a number of bytes sent since last update.

ErrorLog

This class logs internal server errors. The error log includes the error stack. Class ErrorLog provides information about the following announcements: ResponderNotFound, RequestStatusLineTooLarge, ConnectionFailed, ResponseSendingFailed, ListenerFailed, RequestExecutionFailed.

The error event output is written in the following format:

```
[date] remotehost "request" error
```

Where:

[date]

Date and time of the request. %d/%b/%Y:%H:%M:%S %z

remotehost

Remote hostname or IP address, if a DNS hostname is not available.

"request"

The request line exactly as it came from the client.

error

the announcement error description.

For example:

```
[10/Sep/2013:15:41:44 -0400] 127.0.0.1:8000 "GET /error HTTP/1.1"
 RequestExecutionFailed: Responder error
---------- Start Exception stack ----------
SiouX.Hello(Object)>>error:
```

Adding a Logger to a Server

A SiouX server can have several distinct loggers at the same time. The method Server>>logs returns the collection of loggers currently set on a server. By default, the collection is empty.

To turn on logging, use: Server>>logging: true. By default, turning on logging adds a CommonLog to the server and starts logging. Note that #logging: starts or stops all loggers that belong to the server.

To start the default logger:

```
server := Server id: 'StreamLoggerTest'.
listener := server listenOn: 8000 for: HttpConnection.
server addResponder: Hello new.
server logging: true.
server start.
…
server logging: false.
```

To add a new logger, use: `Server>>addLog:`.

To remove a logger, use: `Server>>removeLog:`.

For example, to add a logger to a server:

```
logger := ErrorLog fileName: 'MyServerErrors.log'.
server addLog: logger.
logger start.
```

To change the default behavior, use `Server class>>logFactory:`, e.g.:

```
Server logFactory: [:server | ErrorLog log: server].
```

Alternately, the logger doesn't need to be included in Server `logs` collection. It can be created and run as a standalone logger object:

```
logger := StatusLog
    announcers: server
    frequency: 10 minutes.
logger file: 'MyServerStatus.log'.
logger start.
```

Creating a Custom Logger Class

To create a new logger class, you should subclass either `AnnouncementLogger` or `StreamLogger` and implement the `#log:` method. All received announcements are processed here. The Logger has to specify a set of announcers via the `#announcers:` method and a set of announcements (`#announcements:`) to which it subscribes. By default, all announcements are logged.

As different announcements can have widely different shapes, concrete loggers will likely want to employ a double dispatch against the announcement object to transform the announcement into a form suitable for its specific logging target (be it a file, an internal log structure, a database, a network logging target, a statistics collector, etc).

Note that announcements can be volatile, i.e. they may carry parameters that can change after the announcement is delivered, so in some cases it may be desirable to take a suitable snapshot of the announcement state, rather than simply retaining the announcement instance itself.

All invocations of the `#log:` method are serialized with an internal access lock, which is usually desirable because most logging targets need to be protected against concurrent access. This also usually defines suitable transactional boundaries for the logging action (e.g. to make sure that parts of two separate log records do not end up being interleaved).

However, care should be taken designing the logging action to keep the critical section minimal, as the announcements can be delivered by processes where too large a delay in the logging activity can be detrimental. If the logging action is particularly expensive, it may be worthwhile to offload at least some of it to an additional background process managed by the logger.

Chapter

11

Deployment

Topics

SiouX has been designed to simplify both the development and deployment of Web applications. During development, you may use the Web Server Configuration Tool to manipulate servers, responders, filters, and so forth. Since the Configuration Tool is not available when deploying a headless image, you may load and use the AppeX Server Monitor for observing and changing server configuration paramaters at runtime, via a web interface.

In a production environment, you may also wish to deploy a SiouX application in a headless image that does not require any manual configuration. In this case, you can use an XML configuration file. When using a configuration file, all server-specific parameters can be saved from a development image, and then loaded to create and start SiouX servers in a headless image at start-up time.

Preparing an Application for Deployment

VisualWorks provides a variety of options for application deployment, either building up an image by loading parcels, or stripping a development image using the Runtime Packager (for details, refer to the "Deployment" chapter in the *Application Developer's Guide*).

For a production environment, a SiouX application is typically saved in as a headless image (e.g., select **Save Headless As...** from the **File** menu in the Launcher window), and then started from the command line on the server machine. This is the simplest option for deploying an application.

Any servers running when the headless image is created will be running when it is launched in the production environment. Alternately, you can use a configuration file to start or stop servers in the headless image, as described in a subsequent topic, below. For applications in a production environment, a configuration file is the preferred method of starting servers in a headless image.

For more control over a running web application, it is also possible to use the AppeX Server Monitor tool (for details on the Server Monitor, see the *Web Application Developer's Guide*). If you prefer not to load AppeX, and desire only a minimal responder interface to exit a running image, you may alternately use class ImageQuitter.

Configuring a Responder to Exit a Running Image

In lieu of using the more capable Server Monitor, you can configure a simple responder to exit a running SiouX image. For this, you can use the ImageQuitter responder in the SiouX-Examples package. In this way, you can deploy a minimally-configured SiouX application without also loading AppeX.

To configure the ImageQuitter:

1. Ensure that the SiouX-Examples package has been loaded.
2. In the Configuration Tool, select the desired server and pick **Create ...** from the **Responders** menu. When prompted for the name of a responder class, enter ImageQuitter and click **OK**.

By default, this responder may be accessed at:

```
localhost:8888/image-quitter
```

The port number depends upon your server configuration, of course.

The ImageQuitter is protected by a BasicAuthenticationFilter, whose credentials are **admin** and **password**. These may be changed by modifying ImageQuitter>>initialize.

3. Save the image with the ImageQuitter configured.

Changing the Default Image Start-Up Behavior

By default, SiouX will automatically restart any running servers when the VisualWorks image resumes from a snapshot or is launched headless. This behavior is governed by the shared variable ServerSystem.AutoRestart. On image shutdown, SiouX uses class ServerSystem (a subclass of Subsystem) to stop any running servers, and also to activate them when the image starts (or resumes from a snapshot).

By default, the variable ServerSystem.AutoRestart is set to true (i.e., running servers should be automatically restarted). To change the default value, use this API method:

```
ServerSystem class>>autoRestart: aBoolean
```

This option may be useful in conjunction with a configuration file.

Deploying an Application using a Configuration File

Configuration files are useful in situations in which the specific deployment parameters for your application need to be independent of the application code.

A server configuration file can be created with the Web Server Configuration Tool. (To use this, you must first load the SiouX-Tools parcel.)

Creating a Configuration File

If you have a registered server, you can open the Configuration tool and use the **Servers > Save...** or **Servers > Save All...** menu options to create a configuration file. The **Servers > Save...** option creates a configuration file for the currently-selected server, while **Save All...** creates a single configuration file for all servers in the global registry.

To create a configuration file programmatically, you can use the `Server` class-side methods from the `#persistence` category.

For example, to generate an XML element that describes the configuration of the server `'Examples'`:

```
(Server id: 'Examples') asXmlNode.
```

This returns the following XML Element:

```
<server class="vw.sioux.Server" debugging="false" id="Examples">
 <listener class="vw.sioux.HttpListener" hostAddress="0.0.0.0" port="8888"
 running="true" />
 <responder class="vw.sioux.Hello" path="/hello" />
 <responder class="vw.sioux.WelcomeResponder" path="/" />
</server>
```

To write the server configuration for `'Examples'` to a stream:

```
(Server id: 'Examples') writeOn: aStream.
```

To write the configurations for all registered servers to a default file:

```
Server save.
```

The default name for the file is `WebServerConfiguration.xml`, though this can be changed using the following class-side method:

```
Server class>>configurationFileName: aString
```

Loading a Configuration File from the Command Line

To load a configuration file from the command line, you can use the -wwwconfig flag to specify the file name, e.g.:

```
../bin/win/visual.exe myvisual.im -wwwconfig "WebServerConfiguration.xml"
```

This is the preferred method of configuring a headless image, rather than by saving an image with pre-configured servers.

Creating Servers from a Configuration File

To load a configuration from a file into a running image, select the **Servers > Load**... menu option in the Configuration Tool, and specify a configuration file in the dialog. Loading the configuration file creates and registers the servers. The servers may be started, depending on the condition of their #running option(s).

To load configuration files programmatically, use the API methods in the Server class-side category #persistence.

For example, to load the configuration from the default file (WebServerConfiguration.xml), evaluate:

```
Server load
```

To load the configuration from a specific file:

```
Server loadFrom: aFilename
```

Customizing a Configuration File

If you add new options to your custom server, responder or filter class, you also need to be able to save and re-load these options in a configuration file. This can be done by adding an #asXmlNode method to your custom class (to save the new options), and by changing the #importSnapshot: method, to load and apply the saved options.

For example, let's say you have created a custom Responder class with the new instance variable #deployment. To save this variable in

the configuration file, the responder will require the following two changes. First, to save the new option:

```
TestResponder>>asXmlNode
 root := super asXmlNode.
 root addAttribute:
  (XML.Attribute name: 'deployment' value: deployment).
 ^root.
```

Similarly, to load the saved options:

```
TestResponder>>importSnapshot: aNode
 super importSnapshot: aNode.
 deployment := aNode valueOfAttribute: 'deployment' ifAbsent: [nil].
```

Index