

Two teal-colored geometric shapes: a large square and a smaller square positioned to its upper right, creating a stepped effect.

Web Application Developer's Guide

VisualWorks 8.3

P46-0152-03

Notice

Copyright © 1993-2017 by Cincom Systems, Inc.

All rights reserved.

This product contains copyrighted third-party software.

Part Number: P46-0152-03

Software Release: 8.3

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

CINCOM, CINCOM SYSTEMS, and the Cincom logo are registered trademarks of Cincom Systems, Inc. VisualWorks is a trademark of Cincom Systems, Inc., its subsidiaries, or successors and is registered in the United States and other countries. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation © 1993-2017 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

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	vii
Audience	vii
Conventions	vii
Typographic Conventions	vii
Special Symbols	viii
Mouse Buttons and Menus	viii
Getting Help	ix
Commercial Licensees	ix
Personal-Use Licensees	x
Online Help	xi
Additional Sources of Information	xi
 Chapter 1: Overview	 1
Design Features	2
Architecture	4
Components	4
Loading AppEx	5
Example: "Hello World!"	5
 Chapter 2: Framework	 9
Application Model	10
Implementation	10
Application	10
ApplicationClient	12
JavascriptCode	13

JavaScriptObject.....	14
GenericJavaScript.....	14
CoreCode Library.....	15
Generating HTML Documents.....	15
Declaring HTML Document Components.....	15
Using HTML Templates.....	17
Using CSS Themes.....	17
Working with Application Services.....	18
Defining a JavaScript Service.....	21
Defining a JSON Service.....	21
Defining a File Service.....	22
Defining a Server-Sent Event Service.....	23
Defining an Application-Caching Service.....	23
Accessing Request and Response Data.....	24
Custom Service Types.....	25
Chapter 3: JavaScript.....	31
Working with JavaScript.....	32
Writing JavaScript Code.....	32
JavaScript Code Organization.....	33
Third-party JavaScript Libraries.....	36
Generating a Client-side Request Template.....	37
Internationalization.....	38
Usage.....	38
Implementation.....	41
Framework API.....	42
JSFile Framework.....	43
Using Chrome Workspaces with AppEx.....	44
Implementation.....	46
Minification.....	46
Creating Minified JavaScript (Development).....	47
Using Minified JS Code (Production or Development).....	47

Chapter 4: Messages and Events	49
Sending Messages to the Server	50
Server-Sent Events	52
Chapter 5: Sessions	57
Working with Sessions	58
Enabling AppEx Session Management	58
Linking Sessions between Client and Server	59
Defining a Custom Session Key	59
Setting Session Expiration Parameters	60
Exploring Client and Server Session Data	60
Chapter 6: Active Record	63
Using Active Record	64
Active Record and Database Support	67
Creating Custom Services with Active Record	69
Example: A Genealogy Application	71
Chapter 7: Scaffolding	73
Overview	74
Application Design with Scaffolding	75
Creating an Application	76
Creating an Application using Class WebApplicationBuilder	77
Using the Create Web Application Tool	79
Framework	88
Presenters	88
Domain Objects	91
Page Flow	91
Example: A Genealogy Application	92
Preliminaries	93
Creating Class GenealogyClient	93

Creating Labels.....	94
Working with Scaffolding.....	95
Creating Custom Presenter Classes.....	95
Implementing the Delete Function.....	97
Customizing the Presentation of an Attribute.....	98
Creating Domain Object Relationships.....	99
Validation with Custom EditPresenters.....	99
Client-Side Validation in Scaffolding.....	99
 Chapter 8: Server Monitor.....	103
Using the Server Monitor.....	104
Creating a New Server.....	104
Removing a Server.....	105
Adding a Preconfigured Server.....	105
Accessing your Application from the Server Monitor.....	106
Setting the Responders' Order.....	107

About this Book

This document, the *Web Application Developer's Guide* has been written to help VisualWorks developers create Web applications effectively using the Appex framework.

This guide accompanies the [Web Server Developer's Guide](#), and should enable you to quickly build applications using Appex.

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, browsers, HTML, and JavaScript.

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
<i>template</i>	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	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.
<Select> button	
<Operate> menu	Indicates two keys that must be pressed simultaneously.
<Control>-<g>	
<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 <i>operations</i> that are appropriate for the current view or selection. The menu that is displayed is referred to as the <i><Operate> menu</i> .
<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 <i><Window> menu</i> .

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

- [Design Features](#)
- [Architecture](#)
- [Components](#)
- [Loading Appex](#)
- [Example: "Hello World!"](#)

Appex is a full-featured web application framework and tool set for developers building modern web applications. Appex provides an application model and dedicated tooling to coordinate server-side Smalltalk with client-side JavaScript.

Appex is built using SiouX, the general purpose web server for VisualWorks, such that features of SiouX are also available for developers using Appex.

Design Features

AppeX is designed to address important developments in the evolution of web application design over the past decade. As web applications have become richer and more interactive, they have drawn increasingly on the resources of client-side scripting. Consequently, they have evolved into a more heterogeneous form that combines several distinct languages and runtime environments.

To address these contemporary requirements, AppeX provides support for:

- Rich applications combining Smalltalk with JavaScript
- Library-neutral JavaScript; all popular libraries may be used
- JavaScript support in the VisualWorks toolset
- Framework for building Smalltalk request handlers
- HTML pragmas and templating
- Automatic web session management
- Asynchronous client/server AJAX communication
- Server to client push notification using HTML5 EventSource
- Support for WebSockets
- Support for Active Record

At present, a common approach to application design is to serve web clients a skeletal HTML document that references its static resources (i.e., CSS and JavaScript), perhaps some images, and then runs JavaScript code that does everything else in the browser. Further requests to the server are for single elements of content, which the client-side JavaScript uses to update the document in the browser.

This design approach is supported by a large collection of open-source JavaScript libraries, which provide rich client behavior (e.g., jQuery, Mootools, Prototype, etc.). These libraries have evolved rapidly to enjoy widespread use. Similarly, a growing number of web applications are now designed for off-line browsing, even downloading the entire content as a single HTML file and caching it on the client. AppeX enables you to leverage all of these developments while building your own applications.

For server-side code, developers use the Smalltalk classes they are already familiar with. Writing web request-handling methods is as simple as tagging a method with a pragma declaring the path to the request, while the framework takes care of the rest, including session management and event push notifications. The request handlers can serve various content types, from plain text, to JSON-encoded binary data, to the content of both static and dynamically-generated files.

For client-side code, developers can write object-oriented JavaScript code directly in the VisualWorks System Browser, and the AppeX framework pushes the resulting JavaScript code library to the client's web browser at runtime. Unlike some web development tools, AppeX does not use a mapping mechanism whereby you would write code in Smalltalk and let the framework translate it to JavaScript. Instead, the client code is written in JavaScript, and the code you see in the VisualWorks IDE is what you get in the client web browser.

In this way, AppeX is completely library-neutral; there are no requirements to use any particular JavaScript library. Rather, it enables you to use whichever library you wish. Developers who prefer direct manipulation of the client DOM can do so right out of the box.

Application developers who prefer to work with more friendly DOM API libraries (jQuery, Mootools, Twitter Bootstrap, etc.) can easily include them if they wish, and write the JavaScript code using the library of their choice. One of our goals is to provide web developers with the ability to write web applications in the scripting language and API with which they are already familiar, while integrating them with the rich set of tools included in the VisualWorks IDE.

AppeX includes numerous examples of simple web applications, with and without the use of established JavaScript libraries such as jQuery, as well as an example of serving a completely JavaScript-free, static HTML web site.

Architecture

An Appex web application typically consists of at least two classes: a Smalltalk server class that runs in an image on a server host, and a JavaScript client class which, although written in the Smalltalk IDE, is used to generate JavaScript code that is downloaded and executed by a web browser on a client machine. The two key classes that form the Appex web application framework are `Appex.Application` and `Appex.ApplicationClient`.

`Application` is an abstract class that implements all the server-side behavior of the web application. Its name should not be confused with the usual meaning of the term "application server" — it only indicates that this class (and its subclasses) implements an application's server-side behavior. When writing an HTML5 application using Appex, developers create a subclass of `Application` to handle client requests, and the corresponding client-side code in a subclass of `ApplicationClient`.

`ApplicationClient` is an abstract class that implements the client-side application behavior. Each concrete subclass that you define contains the JavaScript code required to run in the client's web browser. If an application chooses to serve only static HTML files or requires no client-side JavaScript code, it is not required to have an `ApplicationClient` subclass.

For additional details, see: [Framework](#).

Components

The basic Appex framework is delivered as four parcels in the `www` directory of the standard VisualWorks distribution: `Appex-Server`, `Appex-Client`, `Appex-Support`, and `Appex-Tools`. The first three of these provide the basic Appex functionality, including the HTML5/JavaScript web application framework, built on top of the SiouX HTTP server. These are the only three parcels required for runtime deployment of Appex web applications. (For details on SiouX, see the [Web Server Developer's Guide](#).)

To enable tools support for writing JavaScript in the VisualWorks IDE, load the parcel named `Appex-Tools` parcel. This parcel

automatically loads Appex-Server, Appex-Client, and Appex-Support, as necessary.

A number of other, optional parcels are provided in the standard distribution, including: Appex-GenericJavascript, Appex-Internationalization, Appex-ActiveRecord, Appex-ThirdPartyLibraries, Appex-Examples, Appex-Examples-Mobile and Appex-Examples-SharedLibraries. These provide support for non-JavascriptObject JavaScript, internationalization/localization, the Active Record pattern, using shared variables to specify locations and versions of third-party libraries, and working examples of various types of web applications built with Appex.

Loading Appex

To load Appex into your VisualWorks image for application development, open the Parcel Manager (select **Tools > Parcel Manager** in the VisualWorks Launcher window), select the suggested **Web Development** parcels, and load the Appex-Tools parcel by double-clicking on the desired items in the upper right-hand list.

The Appex parcels are configured using VisualWorks package prerequisites, such that all the correct parcels are automatically loaded when you load an Appex parcel. E.g., by loading Appex-Tools, then Appex-Server, Appex-Client, Appex-Support, and several SiouX parcels are also loaded automatically.

Example: "Hello World!"

To help you get started with Appex, the following paragraphs explain how to write a simple 'Hello World!' web application.

This brief tutorial illustrates the key steps in building a web application, and introduces the behavior of some of the core Appex classes. In the interest of brevity, this example does not illustrate any user interaction or session-dependent behavior. The immediate goal is to show how a web application client/server class pair is created, how the service methods are implemented on the Smalltalk side, and how they are invoked using JavaScript on the client side.

To begin, we create two classes for the example application, by subclassing classes in the AppEx framework.

1. For the first one, subclass Application. Name it HelloWorld:

```
Smalltalk defineClass: #HelloWorld
superclass: #{AppEx.Application}
indexedType: #none
private: false
instanceVariableNames: ''
classInstanceVariableNames: ''
imports: ''
category: ''
```

2. Subclass ApplicationClient, e.g., HelloWorldClient:

```
Smalltalk defineClass: #HelloWorldClient
superclass: #{AppEx.ApplicationClient}
indexedType: #none
private: false
instanceVariableNames: ''
classInstanceVariableNames: ''
imports: ''
category: ''
```

3. Point the System Browser at class HelloWorld, and implement the class-side method applicationClientClass, to simply return class HelloWorldClient, e.g.:

```
applicationClientClass
^HelloWorldClient
```

Optionally, you can implement another class-side method `htmlTitle` to return a `String` that is used as the title of the starting HTML page.

4. Create a simple pragma configuration method for the creation of a server that can accept requests for your web application.

For this, extend class `Sioux.Server` in the package that contains your application classes, with a class-side method like this:

```
myDemoServer: aServer
<server: 'Demo HTTP Server'>
```

```
aServer listenOn: 8888 for: HttpConnection
```

Here, the pragma is: `<server: 'Demo HTTP Server'>`. If port 8888 is already in use, you may need to choose another one.

5. In class `HelloWorld` again, implement a class-side pragma configuration method to define a responder to be added to the server. The responder will dispatch requests with a path matching the one in the `path:` argument of the pragma. Note that the path must start with a forward slash.

helloWorldResponder

```
<server: 'Demo HTTP Server' path: '/hello-world'>
```

6. To start the server, evaluate the following code in a workspace:

```
(SiouxX.Server id: 'Demo HTTP Server') start
```

Alternately, you can use the `<Operate>` context menu in the System Browser to create/destroy a server, or to register/deregister the application with the server. These menu picks are available whenever you select an AppEx pragma method in the browser.

7. At this point, you should be able to access the application from a web browser, e.g.:

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

The application shows the default HTML content inherited from `Application`, but there are no message sends from the client to the server other than the session link establishment and some framework-related messages to register interest in server events.

Next, we can create a service method to return some data from the server to the client, and a client function to send a message to the server, invoking the service method.

8. Implement a class-side service method in `HelloWorld` that returns the 'Hello World' greeting:

getGreeting

```
<plainText: 'getGreeting'>
```

```
^'Hello World!'
```

9. Implement an instance method in class `HelloWorldClient` that builds the full DOM on the client. For this, you write a JavaScript function body (without the `'function'` keyword) in the VisualWorks System Browser:

```
buildHtml() {  
  // Create an HTML paragraph element  
  var element = document.createElement("p");  
  
  // Send a SYNCHRONOUS message named "getGreeting" to the server.  
  // The name of the message is the same as the #plainText: pragma argument in  
  // HelloWorld class >> #getGreeting method.  
  // On the server side, the method name and the pragma argument do not have to  
  // be the same.  
  var response = this.messageToServer("getGreeting", null, {async: false});  
  
  // Retrieve the greeting from the response  
  var greeting = response.object;  
  
  // Create an HTML text node and append it to the paragraph element.  
  element.appendChild(document.createTextNode(greeting));  
  
  // Append the paragraph element to the document body  
  document.body.appendChild(element);  
}
```

The Appex class `JavascriptCompiler` translates this code into an annotated method named `buildHtml` that returns the full string of the function definition.

10. Refresh the browser at <http://localhost:8888/hello-world> to see a Hello World! greeting generated by your application.
11. You can now stop and destroy the demo server:

```
(SiouxX.Server.Registry at: 'Demo HTTP Server') release
```

This concludes the "Hello World!" tutorial. Additional examples can be found in the various Appex-Examples parcels.

Chapter

2

Framework

Topics

- [Application Model](#)
- [Implementation](#)
- [Generating HTML Documents](#)
- [Working with Application Services](#)
- [Accessing Request and Response Data](#)
- [Custom Service Types](#)

The AppeX core classes define the general form and behavior of both the server- and client-side of your application. They provide several different ways to generate XHTML documents (e.g., as components, templates, or services), and services for your client-side code that interoperate smoothly with JavaScript (e.g., via JSON, events, files).

To understand the AppeX application model and how it can simplify your project, we'll begin with these core framework classes.

Application Model

To build an application using AppeX, you begin by subclassing `AppeX.Application`. The top-level resources of your application are provided by methods you add to this subclass, which execute in a Smalltalk image on the server host. These resources include the elements that together form an HTML 'shell' document, as well as CSS definitions and any services provided by the application. AppeX provides a mechanism for automatic registration of service methods in your application class. These are typically invoked using Ajax callbacks from the client.

Client-side behavior in your application is managed by subclassing `AppeX.ApplicationClient` (or if you do not wish to use the AppeX client framework, by subclassing `GenericJavaScript` or `JavaScriptClosure`), and adding methods to the subclass. Generally, you do this if your application needs to manipulate its client-side content after the initial HTML has been loaded, or if it needs to keep this content in sync with the state of the server, i.e., it requires some kind of session management. The client-side behavior consists of one or more JavaScript classes which, although written in the Smalltalk IDE, are actually JavaScript 'class functions' downloaded and executed by a web browser on a client machine.

Implementation

The AppeX framework includes five core classes — `Application`, `ApplicationClient`, `JavaScriptCode`, `GenericJavaScript`, and `JavaScriptObject` — which are described in subsequent topics.

Application

When writing an HTML 5 web application, you create a subclass of `AppeX.Application` to handle server requests. The simplest kind of web application would be a single HTML page. For such applications, you can override the default `htmlDocument` class-side method, or take advantage of AppeX's ability to declare individual methods as HTML components.

Class `AppeX.Application` is an abstract class whose subclasses' instances serve a dual purpose. First, this class belongs to the hierarchy of `SiouxX.HttpResponder`, and thereby inherits its basic dispatching and request/response management behavior. Class `AppeX.Application` further extends the dispatching mechanism by including automatic registration, and routing of individual HTTP requests to access the application services.

Since web applications typically require rich client behavior or automatic management of session data residing in the client, the second purpose of subclassing `AppeX.Application` is to model the session data associated with a particular web session. That is, each instance of your subclass represents a distinct web session (for details, see: [Sessions](#)).

Subclasses of `Application` may also represent application-wide data using shared variables in conjunction with specialized domain objects. This enables all instances (i.e., sessions) to share common resources, such as a database broker.

Creating an Application Class

To begin writing a web application, create a subclass of `AppeX.Application` (e.g., `MyNamespace.MyApplication`), by using the **New Class...** dialog from the **Class** menu of the System Browser. Then, implement the class-side method `applicationClientClass`. The return value should be either `nil` or the subclass of `ApplicationClient` that implements the JavaScript client behavior for your application. If `nil`, no `AppeX` JavaScript code is downloaded by the client. (For details, see: [Creating an Application Client Class](#).)

By default, `AppeX` generates a single HTML document as the application's initial page. You can see the contents of the initial HTML by inspecting:

```
MyApplication htmlDocument
```

A typical HTML document has several sections, or components. Instead of writing an entire HTML document in a single piece, `AppeX` provides an API to manage the components as individual methods annotated with `#head:` and `#body:` pragmas. These are discussed in [Declaring HTML Document Components](#).

In some scenarios, such as porting a legacy application with existing HTML, you may want to simply override the class-side `htmlDocument` method to bypass the AppEx component structure and provide your own HTML content.

If your application provides resources other than the `htmlDocument` (e.g., CSS, other HTML pages, non-AppEx JavaScript, etc.), or if it implements client-side JavaScript behavior, you need to add service methods. For details, see: [Working with Application Services](#).

ApplicationClient

The abstract class `ApplicationClient` implements common behavior such as session management, server event notification, and a high-level API for sending and receiving data between the client and the server. Because AppEx JavaScript is object oriented, your subclasses can reuse the common behavior via inheritance, or override select aspects to provide specialized behavior that is appropriate to your application.

If your web application requires specialized behavior on the client, it may include a subclass of `AppEx.ApplicationClient`. The subclasses' instance methods contain the JavaScript code that runs in the client's web browser.

Alternatively, or in addition to defining a subclass of `ApplicationClient`, you can use any combination of `GenericJavascript` or `JavascriptObject` subclasses. However, class `GenericJavascript` specifically does not include class inheritance, and neither `GenericJavascript` nor `JavascriptObject` includes any of the AppEx client framework functionality provided by class `ApplicationClient`.

Each subclass of `ApplicationClient` must be linked to its corresponding (server) application class. This is accomplished by implementing an `applicationClass` method on the server application's class side. For details, see: [Creating an Application Client Class](#).

In the client's browser, an instance of `ApplicationClient` is represented as an object accessible as `$t.application`. Only a single instance exists in each client window, and it is linked to a unique instance of the `Application` subclass on the server. Thus, a pairing of a JavaScript

object on the client and a Smalltalk object on the server constitutes a single web session.

Creating an Application Client Class

If your application design uses rich client-side behavior such as Ajax, Server-Sent Events (SSE) or WebSockets (WS), you can take advantage of the `Appex.CoreCode.js` JavaScript library.

First, using the System Browser, create a subclass of `Appex.ApplicationClient` (e.g., `MyNamespace.ApplicationClient`).

Next, define a class-side method `applicationClientClass` in your application class, that returns your client class, e.g.:

```
applicationClientClass
^MyApplicationClient
```

This method links the client and application classes together, and ensures that the HTML document generated by the server application includes the necessary `<script>` elements to load both the `Appex.CoreCode.js` library and your client application code (e.g., `MyNamespace.MyApplicationClient.js`).

To see the HTML generated by your application, inspect the following:

```
MyApplication htmlDocument
```

JavascriptCode

Appex provides the ability to seamlessly write, maintain, and output JavaScript inside VisualWorks IDE. For this, the framework uses class `JavascriptCode` to manage the source code as instance-side behavior of its subclasses.

Instances of `JavascriptCode` and its subclasses do not have useful Smalltalk behavior. Thus, they should never be instantiated by users. Annotated methods are used to represent JavaScript functions, which may be sent to a client. The JavaScript source is emitted just as it is written — no special code generation is involved.

JavaScriptObject

A subclass of `Appex.JavaScriptCode`, this class represents JavaScript constructor functions used to generate a 'class' in the client's web browser. The Appex framework translates instance-side methods in `JavaScriptObject` to JavaScript functions and binds them to the constructor prototype on the client.

By implementing subclasses of `JavaScriptObject` and including them in one of the JavaScript libraries downloaded to the client, Appex developers can create object-oriented JavaScript with class inheritance. Effectively, `JavaScriptObject` is equivalent to class `Object`, in the sense that each of its subclasses in the VisualWorks IDE are created as subclasses of JavaScript `Object` on the client.

For details, see: [JavaScript](#).

GenericJavaScript

Class `GenericJavaScript` is a direct subclass of `JavaScriptCode`, and so does not inherit from `JavaScriptObject`. The addition to Appex of `GenericJavaScript`, and its subclass, `JavaScriptClosure`, allows Appex to more easily use and interact with JavaScript code that may have been developed outside of the Appex object-oriented JavaScript paradigm, and allows Appex to more easily produce code that lies outside of that paradigm.

`GenericJavaScript` can be used without Appex `CoreCode JavaScript`. It is not yet supported by `JSFile`. `JavaScriptClosure` subclasses are used to generate JavaScript code that creates closures with bound functions, or methods, but that nevertheless do not fall within the `JavaScriptObject` hierarchy.

In order to achieve this, and also to be compatible with `JSFile`, these subclasses require Appex JavaScript `bindMethod` and `name space` support, and hence require the Appex `CoreCode JavaScript`. The Appex-Examples-`GenericJavaScript` package contains code that demonstrates how to write JavaScript applications that make use of the Appex server-side framework and JavaScript IDE, but that are independent of the Appex JavaScript class inheritance implementation and application client frameworks.

CoreCode Library

The AppeX framework uses class `JavascriptObject` and certain of its' subclasses (e.g. `JavascriptFunction`, `JavascriptArray`, and `JavascriptNamespaces`) to generate the corresponding core JavaScript global objects used to bootstrap client-side functionality.

It is not necessary for your application to reference or instantiate any of these classes. The AppeX framework uses the corresponding `CoreCode` library to automatically render the requisite `<script>` elements in the `<head>` of the shell HTML document when it is sent to a web client. The code generated by the `CoreCode` library is included in the `AppeX.CoreCode.js` library downloaded to the client.

Generating HTML Documents

AppeX provides several different ways to compose and generate HTML 5 documents. Documents may be organized as a set of components generated by methods in your application class, or they may be defined as templates, with tokens generated by your application code. In addition, document elements may be generated using specialized service methods.

Each web application is assumed to have a main HTML document that is the entry point into the application. The HTML document loads the corresponding JavaScript libraries, Cascading Style Sheets, and it runs the scripts that initialize the application. By default, this HTML document is returned from the `htmlDocument` class method of your application server class, and is indicated by the use of the `#html:` pragma with the application's root path (e.g., `<html: ">`).

Declaring HTML Document Components

An HTML document includes two main sections, the `<head>` and `<body>` elements. AppeX provides a simple mechanism that enables you to treat the head and the body as a collection of separate elements returned by individual methods in your application class.

You can declare snippets of HTML code that define these elements by writing class methods with `head:` and `body:` pragmas. The arguments in these pragmas are numbers indicating the relative

position of each element within its parent element. Lower position indicators mean the part will be included closer to the beginning of the element, e.g., the HTML code generated by a method with a `<head: 100>` pragma will be generated before a method with a `<head: 101>` pragma. Naturally, all the content declared with the `#head:` pragma will come before any declared with the `#body:` pragma, regardless of their respective relative positions.

Generally, the HTML part methods using these pragmas return `Strings` that together form the final HTML document. They can also return `nil`, in which case the component is skipped in HTML generation, or an instance of a `FileLibrary` subclass (i.e., a `JavascriptLibrary` or a `CSS Theme`).

Appex generates comments throughout the application's `htmlDocument` that indicate the method and the pragma from which each section of the document originates.

To illustrate, consider two example methods in class `MyApplication`:

htmlHelloWorld

```
<body: 100.0>
^'<h1>Hello, World!</h1>'
```

htmlGoodByeWorld

```
<body: 100.1>
^'<h2>Good Bye, World!</h2>'
```

At runtime, Appex renders these as two elements inside the `<body>` element of the HTML document.

A few points about the use of `#head:` and `#body:` pragmas:

- The methods must return a `String` that is the HTML you want to be included.
- You can put as much or as little HTML code as you wish.
- Non-integer and negative values are allowed.
- The names of the methods themselves are irrelevant, but they must not conflict with the Appex API methods.

For additional examples, browse references to `#head:` and `#body:` selectors (i.e., pick **Senders of Selector ...** from the **Browse** menu in the VisualWorks Launcher window).

Using HTML Templates

For additional flexibility in generating HTML documents, you can treat pieces of HTML as templates. A template can include one or more tokens enclosed in double braces, e.g., `{{myToken}}`. (The Appex templating mechanism will be familiar to developers who have used `mustache.js`. For details, see: <http://mustache.github.io/>).

Tokens may be used when you need a small piece of information that occurs in several components of your HTML document, such as a page title or repeated navigation bars.

To provide the content that replaces a token, add a class-side method in your application class that includes the pragma `htmlToken:`. The pragma argument must be a symbol representing the name of the token to be expanded into the HTML document.

For example, to include a page title in both the `<head>` element and somewhere in the `<body>` element, add a class-side method `htmlTitle` that uses the `#title` token (e.g. in `MyNamespace.MyApplication`):

```
htmlTitle
<htmlToken: #title>
^'My Web Application'
```

For additional examples, browse implementors of `htmlTitle`, senders of `htmlToken:`, or browse the method `Application class>>headStart`.

Using CSS Themes

In Appex, a *theme* is a way of organizing and managing CSS resources based on certain criteria. For example, a set of CSS definitions can be used for a certain client, user, locale, application, and so forth.

Themes are defined by subclassing the abstract class `Appex.Theme`. Each subclass of `Theme` can declare snippets of CSS to be sent to the client. The snippets are identified by use of the `<stylesheet:>` pragma in

the instance methods, with a single argument specifying the relative position of the snippet in the resulting CSS.

To include a default theme in your application, override the `defaultTheme` class-side method to return an instance of the default theme you want to use. This will ensure the CSS for that theme will be linked in your application's html document, in a `<link>` element with the id `'appex-theme-css'`.

Themes can be switched on the fly by the client, by executing the method `JavascriptClient.setTheme(<url>)`, where the `<url>` argument contains the relative path of the theme. Your application must ensure that the theme is accessible by declaring a `css: service` method returning an instance of the desired theme.

To see an example of dynamic theme switching, load the `Appex-Examples-Scaffolding` parcel, and explore class `GenealogyApplication`.

Working with Application Services

Once you have created the server and client classes for your application, you typically create services that a client can access. This is done by annotating methods in your application class with pragmas that declare them as service methods. Each service method can be annotated with one of `GET`, `POST`, `PUT`, or `DELETE` pragmas to indicate which HTTP methods may be used with the service. If none of the HTTP methods are included, the default HTTP `POST` method is assumed.

Different service types are expected to return distinct types of objects, specified by `'content-type'` values in the HTTP response header. Typically, the return value from a service method should be an instance of one of the valid `Sioux` content classes (e.g., `String`, `Filename`, `Xtreams.ReadStream`, `BlockClosure`, etc.), depending upon the pragma. For textual content, the `'charset'` is set to `'utf-8'`.

To declare a service method, use one of the standard pragmas defined by the `Appex` framework, e.g.: `#html`, `#js`, `#plainText`, `#css`, `#json`, `#xml`, `#eventStream`, `#file`, or `#appCache`. The behavior of the standard pragmas is described below. Your application can also define and use custom service types (for details, see: [Custom Service Types](#)).

Service method pragmas include a string argument specifying the path relative to your application's responder path. The service method is invoked when a client-side request method sends a request to this path.

To illustrate, let's say you want to add a JSON service called `my-json-service`. For this, you define a class-side service method that includes the `#json:` pragma, and returns an `Association` to be sent to a client as a JSON-serialized object (e.g. in `MyNamespace.MyApplication`):

```
myJsonService
<json: 'my-json-service'>
^'greeting' -> 'Hello World'
```

The path to the service is `my-json-service`. The client retrieves the JSON data using either the `syncMessageToServer()` or `asyncMessageToServer()` method functions (for details, see: [Sending Messages to the Server](#)). The corresponding JavaScript code for sending a synchronous message to receive the service data would be a method function in `MyApplicationClient`, e.g.:

```
myJsonService() {
  return this.syncMessageToServer("my-json-service");
}
```

As a second example to illustrate a different type of service, let's say you want a URL to return the contents of a file stored on the server machine. For this, define a service method that includes the `#file:` pragma, and returns either a `Filename`, or the contents of the file as a `String` or `ByteArray`, e.g.:

```
myFileService
<file: 'my-file.txt'>
^Filename named: 'somefile.txt'
```

Or:

```
binaryContents
<file: 'my-image.jpg'>
^#[ ... byte representation of my image ... ]
```

The pre-defined AppEx service types include:

html

Declares the contents of a response to be an HTML document. It must be complete, valid HTML including the `<!DOCTYPE html>` element. The 'content-type' header field of the response is set to 'text/html'.

js

Declares the contents of a response to be JavaScript code. Any valid JavaScript can be returned from the method. For details, see: [Defining a JavaScript Service](#).

plainText

Declares the contents of a response to be a plain text. The 'content-type' of the response is 'text/plain'.

css

Declares the contents of a response to be a cascading style sheet definition. The 'content-type' of the response is set to 'text/css'.

json

Declares the contents of a response to be a JSON-serialized object. For details, see: [Defining a JSON Service](#).

xml

Declares the contents of a response to be an XML document. The 'content-type' of the response is set to 'text/xml'.

file

Declares that a service method should return the contents of a file that resides on the server. For details, see: [Defining a File Service](#).

eventStream

Declares the contents of a response to deliver Server-Sent Events (SSE) to the client. For details, see: [Defining a Server-Sent Event Service](#).

appCache

Declares to contents of a response to be a cache manifest for an application that is designed for off-line use. For details, see: [Defining a Server-Sent Event Service](#).

Defining a JavaScript Service

Use the `#js:` pragma to define a service method, whose return value is a String containing JavaScript code. The `'content-type'` header field of the response is set to `'application/javascript'`, and `'charset'` to `'utf-8'`. In addition to the regular SiouX content classes, you can also return an instance of `JavascriptLibrary`.

Defining a JSON Service

A JSON service may be used any time the client needs to retrieve structured data from the server. The AppEx framework on the client can parse the returned data and create a JavaScript object whose structure corresponds to that of the Smalltalk object on the server.

Use the `#json:` pragma in a service method in your application class, whose return value is a JSON-serializable object, e.g.:

```
myJsonService
<json: 'my-json-service'>
^'greeting' -> 'Hello World'
```

The return value can be any object, as long as it can be encoded into a valid JSON string using class `Xstreams.JSON`.

Simple objects such as instances of `CharacterArray`, `Number`, `Boolean` and `UndefinedObject` can be JSON-encoded. A `SequenceableCollection`, `Dictionary` and `Association` can be serialized as long as the object structure is not recursive (i.e., uses reverse-pointers). For the list of JSON-serializable Smalltalk classes, inspect the shared variable `Xstreams.JSONWriteStream.ActionMap`.

By default, a JSON service emits keys according to the shape of each class instance, but you can extend class `Xstream.JSONWriteStream` to add custom rules for encoding new classes in JSON format, or override the `Object>>streamingWriteJSONOn:` method.

Defining a File Service

A file service may be used to deliver static files to a client.

Use the `#file:` pragma to define a service method that sends the contents of a file to the client. The file is expected to reside in the application's `documentRoot` directory. By default, the `documentRoot` of each `Application` subclass is the current working directory of the `VisualWorks` image.

Several options are available. In the simplest case, the argument of the `#file:` pragma specifies the name of the file in the `documentRoot` directory, e.g.:

```
actualFile  
<file: 'actual-file.html'>
```

Here, the `'content-type'` field of the response will be derived from the `'html'` extension, and set to `'text/html'`. This is the simplest, and probably the most common way of delivering static files to a client.

As a second option, the service method's return value can be a `Filename` object. This causes the contents of the file to be delivered in the response to the client. If the service method returns a `Filename` that is not absolute, its path is relative to `documentRoot`. The `'content-type'` of the response is set according to the filename's extension.

The path in the `Filename` may be also different than the argument to the `file:` pragma, e.g.:

```
fileContents  
<file: 'some-file-on-server'>  
^Filename named: 'real-file-on-server.js'
```

In this case, from client's perspective, the file being fetched is named `'some-file-on-server'`, but the actual file delivered is named `'real-file-on-server.js'`. In this case, moreover, the `'content-type'` field of the response is derived from the `'js'` extension of the actual file, and is accordingly set to `'application/javascript'`.

As a third option, the service method can return a `String`. In this case, the return value of the method is sent to the client as if it were the contents of the file. The `'content-type'` of the response to `'text/plain'`, e.g.:

stringContents

```
<file: 'my-file.txt'>
```

```
^This is (not really) the contents of a file named "my-file.txt".
```

Any return value other than a `String`, `ByteArray`, or `Filename` will have no bearing on the contents of the response. In this case, AppEx uses the argument of the `#file:` pragma to determine the name of the file to be delivered to the client.

Defining a Server-Sent Event Service

Use the `#eventStream:` pragma to define a service method, whose return value is a one-argument block that accepts an `EventStream` object. In the block, the application can write data and event information to the `EventStream`, following the conventions of the SSE specification.

The AppEx implementation of Server-Sent Events (SSE) follows the specification for the `EventSource` interface:

<http://www.w3.org/TR/2009/WD-eventsource-20091029/>

For example:

serverSentEvents

```
<eventStream: 'get-events'>
```

```
^[:anEventStream |
```

```
anEventStream
```

```
put: (ServerEvent eventName: 'MyEventName' data: 'some data')]
```

Here, the service method returns a block that writes a single event on `anEventStream`. In most scenarios, however, the application would wait for server events before sending them to the client.

Defining an Application-Caching Service

Use the `#appCache:` pragma to define a service method, whose return value is a `String` containing a cache manifest.

Application caching is useful in mobile applications, where the application itself is downloaded to a mobile device, and only the data is to be downloaded when the mobile application is active on the device.

The Appex implementation of application caching follows the provisional W3C specification for offline web applications:

<http://www.w3.org/TR/2011/WD-html5-20110525/offline.html>

Your application should define an `#appCache: service` method if you override the default implementation of `Appex.Application class>>cacheManifest` with your own method. By default, this method returns `nil`, indicating that the application should not be cached. By overriding this method to return a `String`, your application directs Appex to include a `'manifest'` attribute in the `<html>` tag of any pages sent to the client.

To illustrate, consider a manifest that instructs the client's browser to cache an application's default CSS style, the Appex core, and the application itself:

cacheManifest

```
<appCache: 'manifest.appcache'>
^ '
CACHE MANIFEST
defaultStyle.css Appex.CoreCode.js
MyNamespace.MyApplicationClient.js
# All other resources should be downloaded from the web when available. NETWORK:
*
'
```

When the server receives a request for a cache manifest, the `'content-type'` field of the response is set to `'text/cache-manifest'`.

The Appex-Examples-Mobile parcel includes JQueryMobileCalculator, an example application that uses this caching.

Accessing Request and Response Data

When an HTTP request is dispatched to an application responder, an instance of `Sioux.RequestContext` is stored in the active process environment under the `#currentRequestContext` key. Assuming that all

subsequent processing of that request is done in the same process, an application can access the request context and its associated request and response data.

Class `AppeX.Application` provides both class- and instance-side API methods for accessing request and response data, as follows:

currentRequestContext

Returns a `SiouX.RequestContext` installed in the `Application` class >>
`dispatchRequest`: method.

currentRequest

Returns a `SiouX.HttpRequest` that the application is handling in the currently active process.

currentRequestData

Returns a `SiouX.FormData` as it gets parsed from the current request's payload.

currentRequestArguments

Returns an object parsed from the `arguments` field of the current request data. These `arguments` are submitted by the client as a JSON string, so the object will be either one of the primitive types (e.g., `Number`, `Boolean`, `String`), or a `Dictionary`. The type of object returned depends on the data the client sent in one of the messaging methods (`syncMessageToServer`, `asyncMessageToServer`)

currentRequestPath

Returns a `String` that holds the path of the request relative to the server's root.

currentResponse

Returns a `SiouX.HttpResponse` that will be sent to the client once processing is finished.

Custom Service Types

While the `AppeX` framework provides a rich set of default service types, there may be an occasion on which you need to implement a custom service type. For this, `AppeX` provides an extensible

dispatching mechanism, which allows you to define your own service types.

Message Dispatching

In order to build a custom service type, it is important to understand the mechanism for dispatching web requests in AppeX. If we consider AppeX within the larger context of the SiouX server framework, class `AppeX.Application` is in fact a subclass of `SiouX.HttpResponder` from which it inherits most of the request dispatching behavior. In order to activate a web application in SiouX, a responder object must be registered with a `Server`. For the present discussion, the responder is an instance of a subclass of `AppeX.Application`, and we will accordingly speak of it as a *responder instance*.

Since each distinct web session is represented by an instance of the AppeX application class, we can think of the responder instance as a template for future session instances. However, it has no session data, and its sole purpose is to dispatch requests to its class. (For details on creating and configuring SiouX responders, see the [Web Server Developer's Guide](#).)

When the SiouX framework determines that an HTTP request should be routed to a responder instance, that instance receives a `executeRequestFor:` message. In AppeX, additional dispatching is required, as different objects may be doing the actual work. The `executeRequestFor:` method sends a message `dispatchRequest:` to the class of the receiver.

The `AppeX.Application` class >> `dispatchRequest:` method uses the path components of the current HTTP request and method to look up a service type registered via one of the service type pragmas. If no service method's pragma argument matches the request path, a `responseNotFound` message is sent, resulting in a **404 - Not Found** HTTP response.

The pragmas used in service methods are automatically tracked by the AppeX framework. Every time a service method is added/changed/removed, a class instance variable named `requestActions` is reinitialized with the service type pragmas used in the affected class and its superclass chain up to `AppeX.Application`.

If a pragma is found for the requested path in `requestActions`, the method matching the pragma name is invoked, with the pragma itself in argument. For example, if you use the `#html:` pragma in a service method, your application class will receive an `html:` message. This is why, if you add a custom service type to your application, you must also implement a matching class method. It also allows the framework, and the application itself, to use introspection to look up any metadata concerning the system's capabilities, e.g., returning HTTP status codes such as **404 - Not Found**, or providing redirection. These possibilities, though, are beyond the scope of this discussion.

When a service type method (e.g., `#html:`) is invoked with a pragma in the argument, it sends a `dispatchPragma:with:` message with the pragma in the first argument, and a block in the second argument. Service methods can be associated either with web sessions or the application as a whole. The `dispatchPragma:with:` method uses the class of the service method where the pragma was used to determine the service method receiver — either the class itself, or its metaclass. Thus, if a service method is implemented on the instance side, a corresponding session instance has to be found in the session cache, or a new session instance has to be created. The receiver of the next message is then a session instance. Conversely, if a service method is implemented on the class side, the subsequent handling of the request is done by the class. The receiver of the next message is the class itself.

The receiver is used as an argument in the block passed to `dispatchPragma:with:`, and the service method itself is executed. The value returned from the service method is packaged in an HTTP response and the response is sent back to the client.

Adding Custom Service Types

While Appex provides a rich set of pre-defined service types, your application may require writing a custom service type. The dispatch mechanism (described in [Message Dispatching](#)) readily allows developers to add new application-specific service types.

Your application may define additional service types by overriding the method `Appex.Application class >> serviceTypePragmas`. For this, add a

class-side method to your application class, with the same selector as the pragma name.

For example, suppose that your application needs to provide a web service described by a WSDL document that is also served by the application.

For this, first override the `serviceTypePragmas` method in your application class, e.g.:

serviceTypePragmas

```
^super serviceTypePragmas, #( #wsdl: )
```

Next, implement a `wsdl:` method to return a WSDL document. The method name must be the same as the newly defined service type pragma:

wsdl: aPragma

```
^self
dispatchPragma: aPragma
with: [:receiver |
    receiver doSomethingWith:
        (receiver perform: aPragma selector)].
```

After these steps, you can write service methods with the `#wsdl:` pragma, e.g.:

myWsdl

```
<wsdl: 'get-my-wsdl'>
"return a valid WSDL document here"
```

Although not strictly mandatory, the pattern used in the predefined service type methods should be followed when writing custom service types. We recommend using the `dispatchPragma:with:` method to route the request to the proper receiver and service method. This takes care of dispatching to the correct receiver and creating a new session instance if necessary.

The `doSomethingWith:` message in the `wsdl:` example above indicates that you can choose any method name to package the result of the service method, as long as it doesn't conflict with one of the `Appex` service type methods. The `doSomethingWith:` method should set the

response content and content-type fields with proper values. (See also: [Accessing Request and Response Data](#).)

Chapter

3

JavaScript

Topics

- [Working with JavaScript](#)
- [Internationalization](#)
- [JSFile Framework](#)
- [Minification](#)

Appex enables developers to write of JavaScript code directly in the VisualWorks IDE. In addition to special tooling extentions to the System Browser for support of JavaScript syntax, Appex makes it possible to design JavaScript client code using class inheritance. Support for popular third-party libraries is available using service methods.

Appex also includes support for internationalization/ localization of your applications, minification of JavaScript, and interoperability with third-party tools.

Working with JavaScript

The following topics discuss how JavaScript code is organized to leverage the inheritance features of Smalltalk, and how JavaScript functionality is served by AppeX to web clients. Patterns for working with third-party libraries are described, as well as special tooling in the System Browser to simplify writing client-side code that invokes service methods.

Writing JavaScript Code

As part of its core functionality, AppeX includes `AppeX.CoreCode.js`, to extend the client JavaScript with the behavior necessary to support inheritance through a class hierarchy. Based upon this, developers can approach designing their JavaScript client code the same way they would with Smalltalk.

To include client-side code, your application must define a subclass of `AppeX.ApplicationClient`, e.g., `MyNamespace.MyApplicationClient`. Alternatively, if you do not wish to use the AppeX client framework (i.e. not creating a subclass of `ApplicationClient`), you can define one or more classes that inherit from `JavascriptCode` (e.g. `MyNamespace.MyGenericJavascript`). Both `ApplicationClient` and `GenericJavascript` belong to a hierarchy descending from `AppeX.JavascriptCode`, a class with split personality. Its class methods are regular Smalltalk methods responsible for writing out the JavaScript code implemented as instance methods. The underlying behavior of each instance method in `JavascriptCode` (and subclasses) is to return a string that is the source of the JavaScript code as written by the developer. The methods are also annotated to allow the browsing of senders and implementors, just like regular Smalltalk methods.

AppeX, however, does not provide 'static', or 'class' methods for the JavaScript code. JavaScript 'methods' are just functions bound to each class prototype. We could also call them 'method functions', and for the purpose of this document, we use both terms synonymously.

To write a simple JavaScript method, one could enter something like this in the method source editor:

```
add(x, y) {
  var result = x + y;
  this.current = result;
  return result;
}
```

This may not look like a very useful method from the application design perspective, but it illustrates some key points:

- When writing JavaScript methods, do not include the `function` keyword, it is automatically added by the framework when writing out the JavaScript library.
- Each JavaScript method contains just the function declaration (less the `function` keyword), and the body enclosed in braces. The JavaScript code is written 'as is', without the need to follow Smalltalk syntax. In fact, the VisualWorks source code editor provides JavaScript highlighting, autocomplete and basic syntax checking according to JavaScript grammar rules.
- An instance of your JavaScript class is accessed via this keyword, corresponding to Smalltalk `self`.

JavaScript Code Organization

In the VisualWorks IDE, JavaScript is organized in a way similar to Smalltalk. JavaScript functions are represented as 'method functions' in your application client class, which inherits via the class hierarchy through `JavascriptObject` (so that if you are using a subclass of `ApplicationClient`, you also inherit its methods). These JavaScript methods can only be written as instance-side methods, and they can be categorized by protocol just as Smalltalk methods are. While the AppeX framework classes belong to the `AppeX.*` name space, subclasses of `JavascriptCode` belong to the `Smalltalk.*` name space. As an application developer, you decide whether to define your own name space for your application classes, or simply use `Smalltalk.*`. Assuming you define one (e.g., `MyNamespace.*`) and your classes are declared there, they can be accessed in Smalltalk just like other classes (e.g., `MyNamespace.MyApplicationClient`).

When an application class (e.g., `MyApplication`) is linked to its client counterpart via the `#applicationClientClass` method (e.g., by returning `MyApplicationClient`), AppEx includes two `<script>` elements in its HTML document: one for the core AppEx library `AppEx.CoreCode.js`, and one for the application itself. By default, the name of that library is derived from the application client class.

As an example, `MyNamespace.MyApplicationClient` would produce a JavaScript library named `MyNamespace.MyApplicationClient.js`. Using the example class names appearing throughout this document, you can evaluate the following code to see the HTML code AppEx generates and the corresponding library.

```
MyApplication htmlDocument inspect.
```

```
CoreCode javascriptLibrary code inspect.
```

```
MyApplicationClient javascriptLibrary code inspect.
```

Once downloaded by the client, the `AppEx.CoreCode.js` library defines a small number of global variables, all starting with `$`: `$replace`, `$debug`, and `$t`. The last of them, `$t`, holds an instance of the JavaScript `Namespace` class (also defined in the core library), which is similar to the `Root.*` name space in Smalltalk. Conversely, `$t.AppEx` holds a JavaScript `Namespace` with all the AppEx JavaScript classes, and if your application client is defined in `MyNamespace.*` in Smalltalk, `$t.MyNamespace` will hold all your JavaScript classes.

Using developer tools in modern browsers, these objects can be accessed and inspected from the console once the JavaScript libraries have been loaded by the client.

Lastly, after all the JavaScript code has been loaded by the client, a single instance of your application client class is defined as `$t.application`. From within the application itself, this object can be accessed via this keyword, but from the rest of your JavaScript library, you must use `$t.application` to access it.

By providing a JavaScript `Namespace` class modeled after the Smalltalk `Namespace`, and a single global `$t`, AppEx enables access to an entire class hierarchy and a network of JavaScript objects.

To see the JavaScript code organization on the client, access any of the example applications from your web browser, open the browser console, and type the following:

```
$t
$.Appex
$.application
```

If you are not using a subclass of `ApplicationClient`, in order to deliver to the web client the JavaScript code corresponding to the subclasses of `JavaScriptCode` that you have created, you must additionally define a method that identifies to the Appex framework the class(es) that are to be used to generate the client JavaScript code for the library. This should be a class-side method on `Appex.JavascriptLibrary` that has the form:

myArbitraryMethodName

"The pragma identifies the name of the application library.

By convention (default) the name will be the fully qualified name of the application with

.js appended "

<library: 'MyNamespace.MyApplication.js'>

"Add one or more JavaScriptCode subclasses that will be used to generate the library code."

^self new

addClass: MyGenericJavascript;

addClass: MyGenericJavascript2;

yourself.

Just as with class `ApplicationClient`, so too when using `GenericJavascript` you can evaluate the following code to see the HTML code Appex generates and the corresponding library:

```
MyApplication htmlDocument inspect.
MyApplication applicationLibrary code inspect.
```

Note that if you are not subclassing `ApplicationClient`, and are not serving the `CoreCode` library to the browser, then it is your responsibility as the application developer to define and code any relationships between JavaScript Objects on the browser. In other words, only your `Generic JavaScript` will be delivered to the

browser, and thus none of the built-in AppEx client functionality described above will be automatically available.

Third-party JavaScript Libraries

As discussed in the topic [Declaring HTML Document Components](#), reusing existing JavaScript libraries is quite simple. Depending on the library, you must declare an HTML component that contains the `<script>` element linking to an external URL that loads the library. In some cases, libraries consist of both JavaScript and accompanying style sheet definitions. In such cases, you can use `<script>` and `<link>` elements in the same component declaration.

For example:

Use the System Browser to declare an HTML component that should be included in the `<head>` element of the document. It instructs the web browser to load jQuery and jQueryUI. This is done by implementing a class-side method in your Application class, (e.g. `MyNamespace.MyApplication`):

htmlLoadjQuery

```
<head: 100>
^
<link rel="stylesheet" href="//ajax.googleapis.com/ajax/libs/jqueryui/1.10.4/themes/
humanity/jquery-ui.css" />
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>
<script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.10.4/jquery-ui.min.js"></
script>
'
```

In some cases, you may want to choose to serve a non-AppEx JavaScript file from your application, e.g., some legacy JavaScript code implemented in several `*.js` files. In this case, in addition to adding HTML components such as those in the example above, you can add service methods to access the contents of the files.

Example:

Add a component to the `<body>` element of your HTML that will instruct the browser to load a style sheet and a JavaScript file. The style sheet is returned by a service method. A second service

method reads a JavaScript file stored on the server in your application's documentRoot directory.

First, write your HTML component as a class-side method in your Application class:

```
htmlLoadJQuery
<body: 100>
^ '
<link rel="stylesheet" href="my-style-sheet.css" />
<script src="my-javascript.js"></script>
'
```

Next, write a service method that returns the contents of your style sheet (again, as a class-side method):

```
myStyleSheet
<css: 'my-style-sheet.css'>
^ '/* Contents of your style sheet goes here */'
```

Lastly, write a service method that returns the contents of your JavaScript file:

```
myJavaScript
"By returning self, this method instructs AppEx
to look for a file named my-javascript.js
in the application's documentRoot."
<file: 'my-javascript.js'>
```

Generating a Client-side Request Template

AppEx provides custom tooling to simplify the task of writing client-side JavaScript using the System Browser. When a service method is selected in your application class, you can generate a corresponding method function in your application client class. This method function will have the same name as the service method in the application class, and perform an asynchronous message send from the client to the server (i.e., Ajax) with proper arguments. For this, select **Generate MyApplicationClient.myService** from the browser's **Method** menu.

Note that, the generated function is only a template. You still need to write your own `onSuccess` and `onError` handlers. You can comment out or delete the (default) asynchronous message send, use a synchronous send, or rewrite the method completely.

Re-generating the method overwrites any changes you have made manually. When generating client-side request templates using this option, make sure to read the comments in the resulting method functions.

To provide this behavior in the System Browser, Appex keeps track of all pragmas used in the service methods of your application classes, and caches them in the `requestActions` class instance variable. An extension of `BrowserNavigator` in the Appex-Tools parcel uses the data stored in `requestActions` and selectively adds the **Method** menu item for generating the JavaScript template code.

Internationalization

The package Appex-Internationalization extends Appex with a client-side JavaScript/HTML5 Internationalization/Localization framework.

(To load this support into your development image, use the **System > Load Parcels Named...** dialog in the Launcher window, select Appex-Internationalization, and click **OK**.)

Usage

To make use of internationalization support, you call a function in place of hardcoded display strings in your application's JavaScript code.

The function call is of the form:

```
_translate("Key", "DefaultValue")
```

Where "Key" is the string you want to translate to a display value. "DefaultValue" is optional, but if provided, it has two effects:

1. DefaultValue is used for display if the Key is not present in the current locale catalog.

2. `DefaultValue` is used to provide the value of the key-value pair during automatic catalog generation, as described below.

At runtime, the web client downloads the locale (e.g. English, French, etc.) catalogs, and uses them when `_translate()` seeks the appropriate display value for the `Key` provided.

Internationalizing/Localizing your Application

The general steps to internationalize/localize your application are:

- Replace each literal string in your `ApplicationClient` code with a call to `_translate()`. This effects the lookup of a locale-specific string at runtime.
- Create the message catalogs and save them to the appropriate directories.

You can perform and re-perform these steps in either order, and you can build the catalogs incrementally as you add client code to your web application.

Generating Message Catalogs

AppEx includes specialized code tools for generating message catalogs. To use these, you need to load the package `AppEx-Tools` (select **System** > **Load Parcels Named...** in the Launcher window, and select the parcels with the dialog).

To aid in creating a template locale-specific catalog, select a subclass of `Application` in the System Browser and pick **Generate Catalog** from the <Operate> menu in the class pane of the System Browser.

This action can also be performed using code, by evaluating the method `#generateUserMessageCatalogs` against the appropriate `Application` subclass. An illustration with the `HelloLocalized` example is detailed below.

Note that catalogs are generated on a per-application basis. `Application classesForCatalog` answers the collection of classes which are scanned to produce the user message catalogs for an application (by default the `ApplicationClient` class), and can be overridden in subclasses as necessary.

Using Message Catalogs at Runtime

To fetch the appropriate catalog at runtime, the web client can send a request for `_appex_catalogForLanguage` to the server, providing the desired locale as a string argument.

The server returns the appropriate catalog, which the client caches in memory and uses as the localization catalog for future client-side calls to `_translate()`. Alternatively, the web client can implement the Application class-side method `#userMessageCatalog` as:

```
userMessageCatalog
<head: 9999.25>
^super userMessageCatalog
```

This tells the application to insert an appropriate HTML tag into the HTML request for the Application's catalog resource (which will have a name of the form `MyApplicationName._catalogsScript.js`).

The `HelloLocalized` example (discussed below) demonstrates how catalogs can be cached in session storage when using session-based applications, and how a locale can be set in the server session of the Application.

Example: HelloLocalized

As an example of automatically generating a catalog, you can load the `HelloLocalized` application in the `Appex-Examples` package (this is a parcel in the standard distribution).

With the `Appex-Examples` parcel loaded, evaluate the following in a Workspace:

```
HelloLocalized generateUserMessageCatalog.
```

This generates the file: `$VISUALWORKS\messages\visualworks\ en \HelloLocalized.lbl`.

Evaluating `#generateUserMessageCatalog` searches the associated `ApplicationClient` (i.e. `HelloLocalizedClient`) for references to the function `_translate()`, and generates a catalog file based on all key-value pairs that the references provide.

Note that you can specify additional classes (that is, additional to the class specified as the `applicationClient`) to be included in the search for references to `_translate()` by specifying them in the `HelloLocalized` class-side method `#classesForCatalog`. Also note that the catalog generated is for the current locale of your VisualWorks IDE, which is, in this case, 'en'. Additional files for additional locales can be constructed by hand by following this same approach.

The `HelloLocalized` example application also illustrates how to make use of several other features of the internationalization framework, such as `#setLanguageInServerSession`.

Setting the Location of the Message Catalogs

The message catalogs used by the Internationalization framework are stored on the server's local file system. The **Message Catalogs** page in the Settings Tool (open this from the Launcher window) specifies the location of these catalogs. The default location is `$(VisualWorks)/messages/visualworks`, but this can be changed using the Settings Tool.

Implementation

The internationalization framework extends the functionality of several AppEx classes, as well as adding several new ones. The classes of principal interest are:

Application

Includes class-side extensions for generating and serving user message catalogs.

ApplicationClient

Includes instance-side extensions (JavaScript code to be run on the client) for requesting, caching, and accessing user message catalogs.

UserMessageCatalog

A new class encapsulating the script that initializes the `$t_catalogs` client-side dictionary that contains the downloaded catalogs.

Catalog File Structure

The catalog file (file type `.1b1`) has a key-value structure, as illustrated by the following hypothetical line:

```
currentServerTimels='Current local server time is'
```

The catalog file structure, and the directory structure in which it is places, conforms to the Message Catalog structure as described in the VisualWorks [Internationalization Guide](#). The AppeX catalogs can therefore be indexed and used within Smalltalk code in the VisualWorks IDE.

Framework API

The AppeX Internationalization framework contains a few methods in class `Application` that may be of particular interest to application developers:

`asCatalogsLibraryName`

This method calculates the name of the JavaScript catalog resource (a `UserMessageCatalog`) for an application, and returns it as a `String`.

Normally, developers do not need to override this method or specifically make use of the catalog name, as the framework will manage it internally.

`catalogForLanguage`

A service method that expects POST data. Returns the catalog for a particular language, as specified in the request arguments.

`classesForCatalog`

Returns the `Collection` of classes which will be scanned to produce the user message catalogs for this application. You can override this method in your application to specify more places to look when constructing the catalog.

The following methods (also in class `Application`) describe how catalog files are organized on disk:

`userMessageDirectory`

Returns a `Filename` for the directory that contains the user message catalogs.

userMessagesRootDirectory

Returns a `Filename` for the root directory that contains the user message catalogs.

Finally, there are two primary internationalization framework methods on the client, class `ApplicationClient`:

`_setLanguage(aString, aCallback)`

Set the locale in both the browser and the current session on the server.

`_translate(key, defaultString, catalogId)`

Access the substitution values in the client code. The only required parameter is `key`. Return the catalog value for `key`, if one exists. If no catalog value exists, but a `defaultString` is supplied, return the `defaultString`. If no catalog value exists for `key` and no `defaultString` is supplied, return `key` itself.

JSFile Framework

The JSFile framework enables you to write out the JavaScript code in your `JavaScriptCode` classes to the file system, change it there, and automatically reload and reparse it into the Smalltalk image.

One use for JSFile support is for rapid application development (RAD) of JavaScript with the Chrome development tools. You can of course modify JavaScript in the debugger, and have the changes automatically reflected in the VisualWorks IDE and in the browser client.

Moreover, the externalized JavaScript files (which have a `.js` file extension) can be modified with the third-party editor of your choice. E.g., you can write and save your code using an external editor, and set the Appex JSFile support to automatically parse it back into the methods of the corresponding Appex `JavaScriptCode` objects.

Using Chrome Workspaces with AppeX

Synchronizing AppeX JavaScript with the Chrome developer tools consists of three main steps:

1. Load the package AppeX-Examples.
2. Start up the Examples server, which includes the Counter example application.
3. Synchronize the AppeX JavaScript with the Chrome developer tools.

This third step actually involves three actions.

1. Open Settings Tool from the Launcher window, browse **Web Development > JavaScript**, enable, **Supply JavaScript code from: Script files**, and then **Apply** the change.

Alternatively, you can evaluate:

```
Application codeComposer: JSFileComposer.
```

The default script directory in which AppeX places the JSFiles is: `/image/sioux/appex/scripts`, e.g. `C:\vw8.2\image\sioux\appex\scripts`. (On OS X, this would be `/Users/<yourUserId>/vw8.2/image/sioux/appex/scripts`.)

On the **Web Development** page of the Settings Tool, you can change and **Apply** a different **Script Directory**, or set it programmatically by evaluating, e.g.:

```
Application class>>scriptDirectory: 'myJSFiles'.
```

This first step activates a `JSFileMonitor` daemon in AppeX that monitors the specified directory for any changes, to be parsed and imported into AppeX.

It also tells AppeX to write out any changes to the JavaScript made within the VisualWorks IDE, for access by external tools (e.g., the Chrome Developer tools).

Finally, it causes any Application requested by a web browser to "externalize" its JavaScript by writing it to the file system in the specified directory.

2. Open the Chrome developer tools, and add the **Script Directory** mentioned in step (1) to the Chrome DevTools workspace.

(For detailed instructions on this, see: [Set Up Persistence with DevTools Workspaces](#).)

3. Finally, you need to map the file JavaScript system files to Network Resources.

The recommended way of doing this is to first request the network resources through your Chrome web browser, by opening the URL for your Application, e.g.:

```
http://localhost:8889/appex-counter/Appex.CoreCode.js
```

Note: The externalized JavaScript files corresponding to the JavaScript code for an Appex application may not be present at this point, as they are automatically generated when you first request that application's URL from the browser.

Once the JavaScript files have been generated by Appex in the scripts directory, you can right-click a JavaScript file in the Sources pane of the Chrome Developer tools, Select: **Map to Network Resource...**, and then choose the appropriate URL to establish the mapping.

For example, `Appex.CoreCode.js` might map to the resource `http://localhost:8889/appex-counter/Appex.CoreCode.js`. In general, after you've manually mapped one of the JavaScript files to a resource, the rest will map automatically.

After the mappings have been established, you can set breakpoints in the Chrome debugger and change the code when halted at those changes. When you refresh the browser, those changes are reflected both in the browser and imported into the VisualWorks environment. Conversely, any changes made within the VisualWorks IDE are then written out to the file system, and will be reflected in the Chrome browser upon refresh.

Finally, although changes made externally to Appex JavaScript remain in the file system, they are not automatically "published" to Store (just as changes made to the JavaScript from within the Appex IDE are not automatically published.) Therefore, for any

JavaScript code changes made by external tools that you wish to save to your codebase, you must use the Store tools to publish them to a repository.

Implementation

The JSFile framework is comprised of three classes: JSFileComposer, JSFileMonitor, and JSFileParserActor.

JSFileComposer

Checks for JavaScript (stored on the file system) to return in response to a resource request (typically by a web browser).

If no JavaScript file exists, then the superclass will compose JavaScript from the AppeX classes, and this composer will write the code to the file system for use by external tools, as well as returning the composed code to the requestor.

JSFileMonitor

Uses a Timer to periodically (by default every five seconds) execute a block of code that checks the timestamps of .js files in specified directories. If the files are newer than the `timeLastChecked`, JSFileMonitor parses the code into the AppeX environment.

JSFileParserActor

Reads JavaScript files from the file system and parses them to update (i.e. replace, if there has been a change) the source code of corresponding JavaScript methods in AppeX.

Minification

The minification framework enables you to generate minified code for `JavascriptLibraries` during development. This minified code is cached for each `JavascriptLibrary`, so that it can be transparently delivered to the client at runtime upon request for a JavaScript resource (in production or development environments).

To use this minified JavaScript in production, deliver the production Smalltalk image with the minified code pre-cached.

Note: Code minification and JSFile support cannot be used in conjunction. For details, see: [Using Minified JS Code \(Production or Development\)](#).

Creating Minified JavaScript (Development)

In the Launcher window, select **Tools > Web Development > Cache Minified Mode**. This will generate minified code for all JavaScript methods in your development image.

Alternatively, use a Workspace to evaluate:

```
AppEx.JavascriptObject cacheMinifiedCode.
```

By default, class and variable names are also minified (i.e., obfuscated) during this process. To disable this aspect of minification, first evaluate:

```
JavascriptMinifyingWriter isMinifyingVars: false.
```

If you have generated minified code, but no longer wish to use minification, clear the code cache either from the Launcher window menu **Tools > Web Development > Reset Minified Code Cache**, or by evaluating:

```
AppEx.JavascriptObject resetCodeCache.
```

Using Minified JS Code (Production or Development)

To have a minified version of the JavaScript resources for an application returned to the client (typically, a web browser), use the Settings Tool to enable **Supply JavaScript code from: Image methods** on the **Web Development > JavaScript** page.

Alternatively, use a Workspace to evaluate:

```
Application useImageCode.
```

Note that code minification and JSFile support cannot be used in conjunction, since code minification requires the setting **Supply JavaScript code from: Image**, rather than **Supply JavaScript code from: Script files**.

However, code minification *can* be used in conjunction with the setting **Serve the JavaScript scripts as a combined file** (on the same page of the Settings Tool), by evaluating:

```
Appex.JavascriptWriter serveFilesCombined: true.
```

Example

Use the Launcher window to open the Web Servers Configuration Tool (select: **Tools > Web Development > Servers**), and start the Examples-Appex server.

Next, open a web browser on `/genealogy`. To test minification, inspect the JavaScript delivered to the client for the various Appex resources (e.g. `CoreCode.js`) using the tools built into the web browser.

In the VisualWorks IDE, select **Tools > Web Development > Cache Minified Code**. Refresh the browser, to see that the JavaScript libraries now contain minified code.

Chapter

4

Messages and Events

Topics

- [Sending Messages to the Server](#)
- [Server-Sent Events](#)

The core Appex library implements a framework for sending and receiving messages and event notifications between the client and the server. Messages are sent via an API that uses AJAX, while events are handled using the Server-Sent Events (SSE) protocol. Appex supports automatic push notification of events that occur on the server or in other client sessions, using the JavaScript `EventSource` object.

Sending Messages to the Server

The JavaScript support for sending messages to a server is contained in class `ApplicationClient`. The API methods are:

```
ApplicationClient.syncMessageToServer  
ApplicationClient.asyncMessageToServer  
ApplicationClient.messageToServer
```

You can see each method's implementation by browsing them in the VisualWorks IDE, or by inspecting them from a web browser console:

```
$t.AppeX.ApplicationClient.prototype.syncMessageToServer  
$t.AppeX.ApplicationClient.prototype.asyncMessageToServer  
$t.AppeX.ApplicationClient.prototype.messageToServer
```

The AppeX message sending API is essentially a layer on top of AJAX. It shields application developers from the tedium and complexity of repeatedly setting up AJAX calls, providing all the functionality in simple API methods. The underlying implementation uses AJAX to set up the request, and the semantics and options are similar to those of `XmlHttpRequest`.

Synchronous vs. Asynchronous Messages

By default, the JavaScript `XmlHttpRequest()` function sends messages asynchronously. Asynchronous messages are preferred when the client does not expect an immediate result from the server, or when processing of a message can take considerable amount of time.

Developers are encouraged to use asynchronous messaging because it doesn't affect client responsiveness. However, using asynchronous messaging requires setting up event handlers for different stages of an `XmlHttpRequest`. For this, AppeX provides simplified event handlers in class `ServerResponse`.

When an `ApplicationClient` sends a message to the server via `asyncMessageToServer`, an instance of `ServerResponse` is returned. The class provides two generic callback registration methods, `onSuccess` and `onError`. Both functions can be sent multiple times to a single instance of `ServerResponse`.

The following example shows how to send an asynchronous message to the server and set up callbacks.

```
$t.application.asyncMessageToServer("some-path" [, messageArgs])
.onSuccess(function() { /* do something when successful */ })
.onSuccess(function() { /* another action on success */ })
.onError(function(error) { /* action on error comes here */ });
```

An `ApplicationClient` can also send a message via `syncMessageToServer`. In this case, the value returned is an object decoded (via JSON) from a string put into response by the server.

The following example illustrates how to send a synchronous message to the server:

```
var object = $t.application.syncMessageToServer("some-path" [, messageArgs]);
```

Synchronous messages may be used when the client expects data in a response, and the processing of a message takes a very short time.

In rare cases where the client is interested in the raw response data, it may make sense to use `messageToServer`, a more primitive form of the message sending API. The return value from `messageToServer` is always a `ServerResponse`. The client code may set options in the message that determine the HTTP method, whether the request is synchronous or asynchronous, etc.

This example illustrates how you might to send a basic `messageToServer` from client to server:

```
var data, response;
data = {};
response = $t.application.messageToServer(
    "some-path",
    [ arg1, arg2, ... ],
    {async: true, method: "GET"}
);
```

```
response
.onSuccess(function(){ $replace(data, response.data) })
.onError(function(error) { data.error = error });
```

Server-Sent Events

HTML5 defines a mechanism (previously known as Comet) that significantly simplifies event notification from the server to the client. It is specified as Server-Sent Events (SSE) in a W3C document (for details, see: <http://www.w3.org/TR/eventsource/>).

The JavaScript class providing SSE behavior on the client is `EventSource`, which is supported by all major browsers except Internet Explorer. The `AppEx.CoreCode.js` library provides class `EventSourceEmulator` that gets installed as `EventSource` in the case that a browser doesn't support it.

If an AppEx application is configured to use session management (i.e., it has at least one service method on the instance side), a default `EventSource` object is installed on the client, and a corresponding default service method `getServerEvents` will take care of the event notification.

To take advantage of the default SSE support in AppEx, developers must instruct the application client to register interest in server events. To register the interest in an event on the client, use the `ApplicationClient.onServerEvent()` method, e.g.:

```
$t.application.onServerEvent(
  'MyEvent',
  function(data) {
    "do something with @data"
  }
);
```

After the client has registered interest in an event, whenever the server application announces an `AppEx.ServerEvent` with the corresponding event name, an underlying `EventSource` object will receive a notification and route it to execute the anonymous function as shown in the example above. The function's data argument will be the event data sent by the server in JSON format.

To announce a server event to deliver a notification for the above example, use the `Application >> #postServerEvent` method, e.g.:

```
anApplication postServerEvent:
  (ServerEvent eventName: #MyEvent data: 'event data')
```

When the application client is no longer interested in receiving notifications about a certain event, it should cancel the interest by using the `ApplicationClient.cancelInterestInEvent()` method, as in the following example:

```
$t.application.cancelInterestInEvent('MyEvent');
```

Aside from the default event notification mechanism in which a default `EventSource` is handled by `getServerEvents`, developers may choose to write a more primitive form of SSE. The API will depend on whether or not the application client wants to access an application's session data. Note that in both cases, developers would have to write their own message and event handling callback functions on the client. A detailed discussion of the event handlers API is included in the SSE specification and goes beyond the scope of this document.

Primitive SSE without Session Management

This form does not access the session data, but will work in both session-less and session-aware applications.

On the client, use the raw `EventSource` API, e.g.:

```
var eventSource = new EventSource("my-event-source");
eventSource.onmessage = function(event) { /* set up your event handler here */ };
```

On the server, write a service method that includes an `eventStream` pragma. The method must be implemented as a class method, and must return a block which accepts an instance of `Appex.EventStream` in its single argument. When writing data to the `EventStream`, you can use basic API methods that roughly correspond to the SSE specification.

For example:

```
MyNamespace.MyApplication class >> handleEventStream
```

```
<eventStream: 'my-event-source'>
^[:eventStream |
eventStream
retry: 1000;
eventName: 'MyEvent';
data: 'my event data';
postEvent]
```

Primitive SSE with Session Management

This form only works if your application has enabled session management, i.e., it has at least one service method implemented on the instance side.

On the client, use the `ApplicationClient.createEventSource()` method to create an instance of `EventSource`. This will ensure a correct `sessionId` is included in the resulting HTTP request. For example:

```
var eventSource = $.application.createEventSource("my-event-source");
```

The service method on the client would be similar to the one in the previous example, except it would reside on the instance side. Recall that an instance of an `AppEx.Application` subclass represents a session state. This gives the application the ability to send session-specific event notifications to the client.

Understanding EventStream and SSE Fields

The SSE specification defines several field names that are at the beginning of each line written to the response stream by the server. They have specific meanings and are described in detail by the W3C specification (<http://www.w3.org/TR/eventsource/>).

The `EventStream` methods correspond to the fields from SSE specification as follows:

EventStream Method	SSE Field	Note
<code>eventId:</code>	<code>id:</code>	Sets the event id. Interpretation depends on the application.

EventStream Method	SSE Field	Note
data:	data:	Writes a line to the response stream in the form of 'data: '<your data>CRLF. A single data field to be received by the client.
retry:	retry:	Writes a line to the response stream in the form of 'retry: '<milliseconds>CRLF. Determines the amount of time to elapse before the client tries to reopen an EventSource if the connection gets closed.
eventName:	event:	Writes a line to the response stream in the form of 'event: '<eventName>CRLF. Any data: fields that may follow will be appended to this event's data.
comment:	:	Writes a colon to the response stream, followed by the string in the argument. If the string spans multiple lines, it is split and multiple ':' fields are written to the response stream.
postEvent	n/a	Writes an empty line (CRLF) to the response stream to indicate that the current event data is finished.

The AppEx-Examples parcel includes an example of a very simple application using the primitive form of SSE. It is implemented in the `EventSource` and `EventSourceClient` classes.

Chapter

5

Sessions

Topics

- [Working with Sessions](#)

Managing session state is essential in even relatively simple web applications. Traditionally, sessions have been managed by using cookies, but this approach has a number of disadvantages, and has become quite outdated. HTML5 defines a new object type, `Storage`, which can be used to store objects in a web browser. Web browsers supporting HTML5 define a variable `sessionStorage` to manage transient session data.

Appex uses the `sessionStorage` variable in its session management mechanism. The variable can hold data specific to a single web browser tab or window. This allows complete isolation of session data between different web browser tabs, unlike cookie-based session management in which a single session on the server would be accessible from multiple web browser tabs/windows.

Working with Sessions

In applications that require session management, instances of your Application subclass provide access to the session data. Your AppeX application determines whether to include session management in your client code based on the location of the service methods.

Automatic session management is activated in your application as soon as you add a single service method to the instance side of your Application subclass. An application class can optionally contain an instance of AppeXSessionFilter in its settings. The session filter holds an instance of SessionCache, and may be shared among multiple application classes, thus allowing several applications to share access to a common session state. The general session caching mechanism is described in [Web Server Developer's Guide](#). However, AppeX differs slightly from the way the SiouX responders would typically set up session filters to keep track of sessions. AppeX session management is simpler to activate, and makes the use of session state information directly in the instances of Application subclasses.

Enabling AppeX Session Management

To make a web application session-aware, define a subclass of AppeX.ApplicationClient, and define at least one service method on the instance side of your Application subclass.

For example, let's say that you create MyNamespace.MyApplicationClient (a subclass of AppeX.ApplicationClient), and on the instance side of MyNamespace.MyApplication, you implement a service method using one of the default service type pragmas, e.g.:

MyNamespace.MyApplication >> serviceMethod

```
<plainText: 'example'>  
^'This is just an example'
```

By doing this, AppeX will insert the following snippet of code in your application's HTML document:

```
<script>(new$.MyNamespace.MyApplicationClient()).installSession(!"sessionKey":  
"sessionId");</script>
```

This script instructs the client application to link an instance of `MyApplicationClient` in the web browser to an instance of `MyApplication` on the server, and to use `sessionId` as the name of the form variable to hold the session key value. In this example, subsequent POST requests from the client will include `sessionId` in the form data, with a value assigned by the server. Using this mechanism client and server-side session data can be synchronized.

Linking Sessions between Client and Server

The `AppEx.ApplicationClient.linkSession` method checks if any session information is already stored in the browser's `sessionCache`, and if so, it uses an existing session id to try to re-establish a link. This is useful if the user reloads an open web page, allowing the web application to display content that is appropriate to the last known state of an existing session.

On the server side, if a session with a requested session id already exists in the session cache, its data is used as the session state. Otherwise, a new instance of the application gets created along with a new instance of `SiouX.Session`, and the application instance is used as the session state in the session data.

You can experiment with `sessionStorage` and explore how data on the client and on the server relate to each other. Load your application, or one of the examples provided with `AppEx`.

Defining a Custom Session Key

By default, the session key variable used to pass the session identifier between client and server is `#sessionId`. It is defined in `AppEx.Application` class >> `#sessionKey`. Developers may specify the name of the session key variable they want to use, by overriding the class method `sessionKey` and returning their own variable name, e.g.:

```
MyNamespace.MyApplication class >> #sessionKey
  ^#mySessionId
```

By doing this, the `<script>` snippet in the HTML code will replace `sessionId` with `mySessionId`, instructing the client that message requests should use `mySessionId` as the query variable holding session id.

Setting Session Expiration Parameters

A `Sioux.SessionCache` object can include a `SessionInactivityRule` that specifies the amount of time after which a session expires if it hasn't been touched. `Appex` provides a convenient mechanism to set the session expiration time, using the 'time to live' term. The three methods of interest are `#defaultSessionTTL`, `#sessionTTL`, and `#sessionTTL:`.

Developers can override `Appex.Application` class `>> #defaultSessionTTL` to return a `Duration` that specifies the amount of time a session will be active before expiration. The default value is 10 minutes. Each time a service message is received from the client, if the receiver is a 'session instance' of an `Application` subclass, its corresponding `Session` gets touched and the time is reset.

The `defaultSessionTTL` only affects the session cache at the time of creation — the first time session management is required. If you need to change the session expiration period, use the `sessionTTL:` class method, e.g.:

```
MyApplication sessionTTL: 3 minutes
```

Setting the `sessionTTL` to `Duration zero` or returning `Duration zero` from `defaultSessionTTL` results in removing the `SessionInactivityRule` from a `SessionCache`, effectively keeping the sessions in the cache indefinitely. Since this can result in a serious increase in memory footprint over time, developers should exercise caution. In such a case, you should implement your own session caching rules and make sure that from time to time, unused sessions are purged from the application session cache.

Exploring Client and Server Session Data

The following examples illustrate how developers may inspect session data on both the client and the server. These may be useful during debugging.

To begin, we will use the `Server Monitor` application implemented in the `Appex-ServerMonitor` parcel. Start the `Server Monitor`, and open a web browser by evaluating this code in a `Workspace`:

```
(Sioux.Server id: 'ServerMonitor') start.
```



```
ExternalWebBrowser open: 'http://localhost:8001'.
```

Next, open the web browser console, and type the following:

```
JSON.parse(sessionStorage.appex_session)
```

The browser should display an object that holds information about the last known session for the application. Copy the long string that represents the session ID.

Next, in the VisualWorks IDE, inspect the following:

```
ServerMonitor sessionCache findSession: <the copied session ID>
```

Here, you can see the client and the server representations of a single session link. However, the actual session data is kept in the instances of the application class (in this case, `ServerMonitor`) on the server, and the application client class in the web browser (i.e., `ServerMonitorClient`).

With the inspector open on a `Sioux.Session` in the VisualWorks IDE, drill into the `applications` instance variable of the session. The inspector should show a `Set` holding an instance of `SessionMonitor`. Alternatively, you can inspect something like:

```
(ServerMonitor sessionCache  
 findSession: <the copied session ID>) applications
```

to achieve the same effect.

When inspecting an instance of `SessionMonitor`, two noteworthy instance variables are `session` (it points back to the `Session` instance we just came from) and `serverName`. The latter holds the name of the currently selected server in the application.

On the client side, you can also inspect the session information. In the web browser console, enter:

```
$t.application
```

This should show something like:

```
'ServerMonitorClient {url: "http://localhost:8001/", sessionKey: "sessionId", sessionId:
<session-id-string>, session: Object, eventChannel: Object...}'
```

Expand the object to see the session data on the client. The interesting data element will be `selectedServer`, which should have the same string assigned to `serverName` on the Smalltalk side.

Chapter

6

Active Record

Topics

- [Using Active Record](#)
- [Active Record and Database Support](#)
- [Creating Custom Services with Active Record](#)

Active Record is a pattern for object-relational mapping, first described by Martin Fowler. It makes it easy to persist a domain model, with an absolute minimum of metadata. The pattern uses a set of standard conventions to map domain classes to database tables, so that a single instance of a class represents a row in the corresponding table.

Appex includes support for the Active Record pattern, to simplify the implementation of Single Page Web Applications that use a database for persistent storage. The Appex-ActiveRecord and Appex-Scaffolding parcels contain the classes implementing this behavior.

While the Appex-ActiveRecord package can be used independently of the Scaffolding framework, in most cases developers will want to use them together, taking advantage of the UI rendering capabilities in Appex-Scaffolding.

Using Active Record

Developers can declare service methods to package Active Record objects (i.e., database rows) as JSON-formatted strings in an HTTP response.

To facilitate this, the Active Record framework adds support for the service pragma `#activeRecord:class:` to class `Application`. In many cases, simply using this pragma in a service method is the only thing that a developer needs to do to make use of Active Record.

In situations where a service request should result in a more complex action, the developer can modify the service method to return a block that will get executed during the HTTP request dispatch phase. (For details, see: [Creating Custom Services with Active Record](#).)

Depending on the HTTP request method, the service pragma `#activeRecord:class:` does the following:

GET

This method corresponds to a READ transaction. It finds all active records for a specified path, encodes them as a JSON collection and returns it to a client.

POST

Corresponds to an UPDATE transaction. It finds an active record for a specified path and id, updates the domain object, commits the change to the database and returns the updated record to a client. If an active record is not found by the id an error will be raised. The active record service for POST requires a client to provide an active record `#id` otherwise the server returns an error.

PUT

Corresponds to a CREATE transaction. It creates a new active record in the database and returns the newly created active record object to a client. The active record POST service will raise an error if a client sends an active record with `#id`.

DELETE

Corresponds to a DELETE transaction. It deletes an active record from the database and returns the deleted record to a client. A client has to provide an active record `#id`.

Encoding an Active Record object

In Active Record, there are different encodings for simple attributes and relations. Simple attributes are encoded as key/value pairs. Active Record relations, by contrast, have a special encoding that includes information about the relation path and the relation print presentation. The relation path is used to retrieve the relation object(s) from the server. The encoding can be customized to meet the specific needs of your application.

To illustrate, consider an example of encoding a `Driver` object (from the `Appex-Examples-Scaffolding` package):

```
car1 := Vehicle new
  make: 'Honda';
  model: 'SUV';
  mileage: 1000;
  yourself.

car2 := Vehicle new
  make: 'Chevrolet';
  model: 'Camaro';
  mileage: 1000;
  yourself.

driver1 := Driver new
  name: 'John';
  address: '123 Street';
  isProfessionalDriver: false;
  vehicles: (OrderedCollection with: car1 with: car2);
  yourself.
```

To represent this object, the JSON-encoded string sent to a client would be:

```
{
  "id": 1,
  "name": "John",
  "address": "123 Street",
```

```
"isProfessionalDriver": false,
"#vehicles":
  {"path": "vehicle",
   "relations": [
     {"Honda SUV mileage: 1000": 1},
     {"Chevrolet Camaro mileage: 1000": 2}]
  }
```

The `#vehicles` relation encoding includes the path to retrieve the relation objects and the collection of `#relations`, i.e.:

```
{"Honda SUV mileage: 1000": 1}
```

where:

key

`Vehicle>>webPrint` value (by default it is `#printString`).

value

`Vehicle` id.

Customizing responses from an Active Record service

An incoming request includes mandatory `#input` query parameters for Active Record, and optional `#output` response parameters. If `#output` parameters are not specified, the service method returns a JSON-encoded Active Record collection. If `#output` parameters are specified they can be: `#self`, which returns the active record; or `#xxx`, which executes the `#xxx` method for the Active Record object and returns its result.

For example, a client sends an HTTP request:

```
GET person\arguments
{
  "input": {"id": 1}
  "output": ["#webPrint", "name"],
}
```

specifying `#output` as `#('#self' '#webPrint')` for a `Person` object, will produce the following reply string:

```
[{
  '#self': {'id': 1,
            'name': 'Smith',
            '#address': {
              'path': 'address',
              'relations': [{'1000 Any Street': 1}]}
            },
  '#webPrint': 'Person: Smith'
}]
```

If a request doesn't include `#output` parameters, the reply for a `Person` object is:

```
[
  {
    'id': 1,
    'name': 'Smith',
    '#address': {
      'path': 'address',
      'relations': [{'1000 Any Street': 1}]
    }
  }
]
```

Active Record and Database Support

The Appex implementation of Active Record makes use of the Glorp Object-Relational mapping framework. Generally, the operational details of Glorp are handled by Appex, but your application is responsible for configuring the database connection and some knowledge of Glorp may be helpful when designing your application. For complete details, consult the [Glorp User's Guide](#).

Connecting to the Database

To connect to a database, your application needs to implement a class-side method named `#newActiveRecordLogin` that returns a `Glorp.Login` object. E.g. as a class-side method in `DriverApplication`:

```
newActiveRecordLogin
^Glorp.Login new
  database: Glorp.SQLite3Platform new;
  username: 'example';
  password: String new;
  connectString: 'driverTest.sqlite';
  yourself
```

Obtaining the Database schema

If a client sends a request whose method is `GET /baseURL/initializeActiveRecordApplication`, the Active Record framework creates a `GlorpSession` object, and returns information about the database schema.

For example, in the `DriverApplication`, the `Driver` schema description is sent to the client, encoded like this:

```
{
  "id": {"isNullable": false, "type": "Number"},
  "name": {"type": "String", "size": 20},
  "address": {"type": "String", "size": 40},
  "isProfessionalDriver": {"type": "Boolean"},
  "vehicles": {"isCollection": true, "isNullable": true, "path": "vehicle" }
```

The schema description includes type descriptions for all `Driver` attributes. The line: `'vehicles': {'path': 'vehicle', 'isCollection': true}` represents a Glorp proxy encoding where the `'path': 'vehicle'` is used to retrieve `Vehicle` objects from the server.

To provide information about Active Record relations, your application needs to provide a map for domain classes and path request. This is done by defining a special class-side method in your application, that returns a `Dictionary`.

For example, as a class-side method in `DriverApplication`:

```
newActiveRecordToPathMap
^Dictionary new
  at: Driver put: 'driver';
  at: Vehicle put: 'vehicle';
  yourself
```

Using a Glorp Session

A Glorp session can be specified on the instance or class side of an application. For an example of the latter, see: `Application class >> initializeGlorpSession: aGlorpLogin`.

When it is specified on the class side, the instance of `GlorpSession` is stored in `Application class >> settings`.

Each Active Record service request registers a `GlorpSession` in the `ProcessEnvironment` via the Active Record setter method. This is necessary to use the Active Record API, such as `<aClass> findAll`.

It is up to the application to ensure this method is sent early in a request dispatch if the Active Record API is used. The `#activeRecord: class: service` does this by default, but any other service intending to use Active Record must do this explicitly.

Creating Custom Services with Active Record

When using the Active Record framework, there may be occasions when you want to create your own custom service. Appex provides a way to do this, such that you can build from the simplicity of the Active Record pattern. (For a general discussion of custom services in Appex, see: [Custom Service Types](#).)

To decide if this is necessary for your application, let's briefly recall how requests are dispatched.

In the case of applications that use Active Record, the Appex request dispatcher uses the `#activeRecord: class:` method by default. Note that application services using this method can only handle a single Active Record instance per request. Moreover, depending on the

type of the request (e.g., GET, POST, DELETE, PUT), there is a fair amount of specialized overhead in processing, dispatching, and encoding.

If your application needs to handle active records from different tables in a single request, if there are several `#ids` in a single request (e.g., `#customerID, #orderID, #bankAccountID + data`), or if you only need to send a simple string back to the client (e.g., `'Success'`), you may want to implement a custom service. The most common case would be that your application needs to update several different active records at once, and you want this to occur as a single transaction.

Creating a custom service for your application involves two steps: first, you must declare the service, and second, you need to declare that a specific path uses that service. In the following paragraphs, these steps are described and then illustrated with a concrete example.

1. To declare a service, add a method to your application that includes the `#serviceType`: pragma notation, e.g.:

```
firstMethodName: pragma  
<serviceType: #PUT>
```

Here, recall that the Appex framework creates a service whenever you define a method that includes a pragma (notation) of the form: `<serviceType: #XXX>`, where XXX is GET, PUT, POST or DELETE, one of the HTTP methods. However, simply declaring a service in this way is not enough to allow clients to use it. There must also be a separate method that associates the service with a particular path in the URL.

2. The second step, then, is to create another method that associates the name of the first, new service method with a path, e.g.:

```
secondMethodName  
<firstMethodName: 'path'>
```

Here, the name of the second method is arbitrary, but the name of the first method is located in the pragma, paired with the path via which it is to be accessed.

By defining these two methods, the following execution pathway is established: when a client sends an HTTP PUT request at path, the Appex dispatcher uses the path and HTTP request type to invoke `#firstMethodName: pragma`.

Example: A Genealogy Application

To illustrate this pattern, we can use the `GenealogyApplication` example. This is provided in the `Appex-Examples-Scaffolding` parcel (load this using the Parcel Manager, if necessary).

1. Create a new class-side service method in `GenealogyApplication`, by using the `<serviceType:>` pragma, i.e.:

```
removeParentService: pragma
<serviceType: #DELETE>
```

The argument in the `<serviceType:>` pragma establishes this service and defines its HTTP method as DELETE.

2. Define a class-side method, also in `GenealogyApplication`, method to associate `#removeParentService:` with a specific URL path:

```
removeParentFrom: aChild relation: aString
<removeParentService: 'removeParent'>
```

Again, the pragma selector in this method (`removeParentService:`) is the same as the method selector for the previously declared service. This has the effect of associating the service with the path.

With these two methods defined, an incoming DELETE request with the path `'removeParent'` will be dispatched to `GenealogyApplication` class `>> removeParentService:` with the argument `'pragma'`. This argument can optionally be used to refer back to `#removeParentFrom:relation:`, as is done in the remainder of the code in class `GenealogyApplication`.

Chapter

7

Scaffolding

Topics

- [Overview](#)
- [Application Design with Scaffolding](#)
- [Creating an Application Framework](#)
- [Example: A Genealogy Application](#)
- [Working with Scaffolding](#)

Scaffolding is a framework for building CRUD (Create, Read, Update and Delete) applications that use a database for persistent storage. It provides a simple way of presenting data to a web client, following a very flexible method of object traversal.

Scaffolding is used in conjunction with the Active Record framework to simplify the implementation of Single-page web applications. In this type of application, Scaffolding is responsible for rendering data to the UI, while Active Record takes care of persistence.

You can use Scaffolding either to build an application from scratch, or from an existing database. For the latter case, Scaffolding includes several tools that can help speed and simplify the process.

Scaffolding is recommended for developers who wish to do a minimal amount of JavaScript and HTML coding, since it provides a convenient out-of-the-box framework for domain traversal, display, and CRUD.

Overview

Conceptually, AppeX is a framework for building "single page" applications. As a consequence, switching pages on the client generally does not involve requesting new URLs, but rather involves using client-side JavaScript and Ajax to change the appearance of the page that is displayed in the client's browser. In this sense, the application may be said to appear as a series of "screens" within the client's web browser. To describe the Scaffolding framework, and how you can use it, the topics that follow primarily address the client-side code.

Scaffolding is meant to address a design problem posed by CRUD applications, of (potentially recursive) display and modification of domain objects stored on a server. The framework solves this problem (and the subsidiary problems of validation and data consistency between client and server) with a few distinctive strategies that deserve explanation.

Scaffolding and Active Record

While Scaffolding requires Active Record, you can also build an application using Active Record without invoking Scaffolding at all. While Scaffolding can save you a lot of time, it may be possible to code a simpler application, more customized to your needs than would be possible with Scaffolding. The decision to do this depends on how averse you feel to writing JavaScript and designing page flow. Either way, the Scaffolding framework and examples illustrating its use may serve as a helpful "how to" reference for common design requirements of AppeX-ActiveRecord web applications.

Features and Invariants

The Scaffolding framework encompasses several sub-frameworks, including bread crumbs, theming, and validation (both single-value and page-wide). You can use any or all of these options, mixing and matching, extending or overriding at will. The framework provides built-in default displays, buttons, and highlighted navigation links.

These built-in capabilities come with one large caveat: in order for all these features to work automatically, you must follow the patterns of the Scaffolding framework. Here, it is important to understand that Scaffolding presupposes some general patterns of navigation between objects in the domain hierarchy and the approach to displaying them for reading and updating.

These patterns will become more apparent as you become familiar with the built-in pages, their links and buttons. For the moment, this means that *either* your page flow must follow the general pattern of a Scaffolding application, with rendering handled through the `JQueryRenderer`, *or* you'll need to do some (possibly extensive) additional coding.

Customization and Extensibility

When using the Scaffolding framework to build a web-based user interface, typically you create customized `Presenters` as part of your web application, adding methods that are invoked by the framework. The methods you create are expected to conform to one of several expected patterns, which will be explained in detail shortly.

For brevity, and clarity, the following topics and example application illustrate the patterns of the default Scaffolding implementation and its behavior, including the "Scaffolding Recommended" avenues for GUI customization. The focus is on the major areas of configuration that are used by a typical developer.

With that said, this is Smalltalk, so there is nothing preventing you from implementing your own page flows and renderers. The behavior of the Scaffolding framework is easily extensible and/or configurable, in large part because of the dynamic nature of both Smalltalk and JavaScript.

Application Design with Scaffolding

The design of an application that uses Scaffolding can be divided into two pieces: the server-side service methods implemented in the application class, and the subclasses of `JavaScriptCode` that generate the JavaScript used in the client's browser.

On the server side, you can declare service methods to package Active Record objects (i.e., database rows) as JSON-formatted strings in an HTTP response.

For generating the client-side code, the `JavascriptCode` classes implemented in Smalltalk (e.g., `GenericPresenter` and its subclasses), generally correspond to JavaScript libraries of the same name that are emitted for use in the client's web browser. In fact, the names of the subclasses of `JavascriptCode` that you create as part of your application are used to generate the names of the library files as well as the JavaScript "user-defined classes".

In conceptual terms, this means that class `GenericPresenter`, for example, is in a sense all of the following:

1. A subclass of `JavascriptCode` in Smalltalk, the container (or more technically, producer) for the `GenericPresenter.js` library of code delivered to the client.
2. The source of the user-defined JavaScript class `GenericPresenter` (which would be instantiated on the client by invoking `new GenericPresenter`).
3. A "Generic Presenter" page in the client browser, that represents an instantiation of the class and generates the XHTML that is rendered in the client's browser as a view on a particular domain object.

For further details, see the discussion of [Presenters](#).

Creating an Application

Scaffolding provides two ways to build a web application from an existing database: you can either use class `WebApplicationBuilder` to create it programmatically, or else interactively, using the **Create Web Application** tool.

Regardless of which approach you choose, you must first load the package `Appex-Scaffolding-Tool` (located in the `/www` directory of the VisualWorks installation). The `Appex-Examples-Scaffolding` package also includes code that creates a sample SQLite database.

To illustrate the use of class `WebApplicationBuilder` and the **Create Web Application** tool, we shall use this sample database to build a simple web application for managing a pool of vehicles.

After loading `AppeX-Examples-Scaffolding`, evaluate the following code:

```
AppeX.DriverApplication initializeDatabaseRecords.
```

This creates the sample database in your image directory, entitled `driverTest.sqlite`. It contains `DRIVER` and `VEHICLE` tables, and some example rows.

Note: While this example should work out of the box, you may need to install or configure SQLite. For some quick details, look at the package comment in `AppeX-Examples-Scaffolding`.

Creating an Application using Class `WebApplicationBuilder`

Use class `WebApplicationBuilder` to create an application programmatically.

1. First, create an instance of `WebApplicationBuilder`, e.g., using a Workspace variable:

```
builder := AppeX.WebApplicationBuilder new.
```

2. Next, set the database connect parameters:

```
builder
  activeRecordSystemLogin:
    (Glorp.Login new
      database: Glorp.SQLite3Platform new;
      username: 'example';
      password: String new;
      connectionString: 'driverTest.sqlite';
      yourself).
```

3. Class `WebApplicationBuilder` provides two ways of initializing the Glorp descriptor system:

1. If the application doesn't have a Glorp descriptor system, set the option to create it and create domain classes from tables:

```
builder createNewDescriptor: true.
```

2. If you have already have a Glorp descriptor system specify it, as follows:

```
builder initializeDescriptorSystem:  
myActiveRecordDescriptorSystem.
```

Here, the argument is expected to be a subclass of `Glorp.ActiveRecords.ActiveRecordDescriptorSystem`.

This class can also be created by the Glorp Mapping tool. For details, see the *Glorp Developer's Guide*.

4. Specify the application name. By default, the `WebApplicationBuilder` uses this as the name of the package that will contain the newly-created code, as well as its name space name.

```
builder initializeApplicationNamed: 'DriverTest'.
```

5. If you chose the first option of creating a descriptor system (i.e., `createNewDescriptor: true`), you need to specify the tables that the builder uses to create domain classes.

There are two options for this. In both cases, the tables must include an `#id` column. Use `#selectAllTables` to select all tables in the database or, alternately, `#selectTables:` to specify the tables via an `Array of Strings`.

For this example, let's use the later option:

```
builder selectTables: #('driver' 'vehicle').
```

6. Specify the path of the initial request to be issued by the client when the application gets loaded in the web browser. This path is set in the client JavaScript function `mainPath()`:

```
builder clientMainPath: 'driver'.
```

7. Specify the server port:

```
builder port: 6767.
```

8. Finally, create all classes in the web application:

```
builder createApplication.
```

At this point, the builder creates:

- The `DriverTest` package
- The application class `DriverTest.DriverTestApplication`, which includes services for two tables: `Driver` and `Vehicle`.

Class `DriverTestApplication` provides basic services to display a list of all table records, add a new record, update existing records, and remove records. This can be seen in the implementation of `DriverTestApplication>>processDriver`:

```
<GET>    "Display all Driver table records"
<POST>   "Update a record"
<PUT>    "Add a new record"
<DELETE> "Remove a record"
<activeRecord: 'driver' class: #{Appex.Driver}>
```

9. Finally, to test the application, evaluate:

```
builder testApplication.
```

This opens a default web browser and displays a list of drivers.

Using the Create Web Application Tool

When building your application from a database, you will need domain classes and a Glorp descriptor system class. The tool provides two options:

- Create the domain classes and descriptor system from existing database tables, and then create a web application
- Create a web application from an existing Glorp descriptor system

For the purpose of illustration, we shall describe the steps for both options.

1. To open the tool, select **Tools > Web Development > Create Web Application** in the Launcher window.
2. The **Specify DB Connection** page prompts you for login details, to connect to the database. Provide the required parameters, as shown:

The screenshot shows a dialog box titled "Creating Web Application from Tables". The main heading is "Specify DB connection, then schema (if needed)". The dialog contains the following fields and controls:

- Connection Profile:** A dropdown menu.
- Interface:** A dropdown menu with "SQLite3Connection" selected.
- Environment:** A dropdown menu with "driverTest.sqlite" selected.
- User Name:** A text input field.
- Password:** A text input field.
- Buttons:** "Save...", "Delete", "AdHoc SQL", "Clear", "Get Schemas", "Help...", "Next >", and "Cancel".

Note that SQLite does not require a user name or password. (For details on this interface, consult the documentation for the Glorp Mapping tool.)

Click **Next**.

3. On the **Application Settings** page, you are presented with the two options described above: (1) create the domain classes and descriptor system from the database, or (2) use an existing descriptor system class. For the first option, we can specify the following values:

Creating Web Application from Tables

Application Settings

Application name:

☒ Create descriptor system:
☐ Use existing descriptor system:

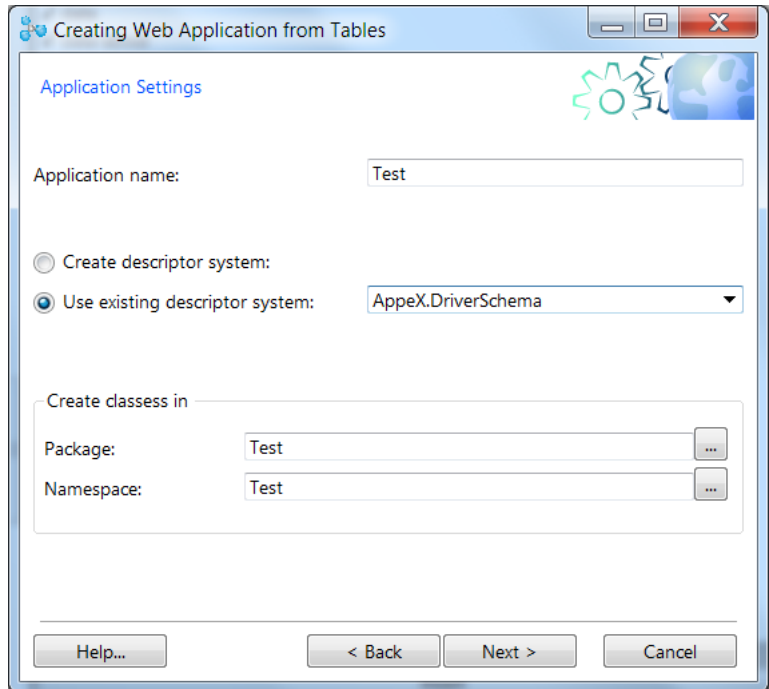
Create classess in

Package: ...

Namespace: ...

Help... < Back Next > Cancel

Alternately, we could choose the second option, specifying class `Appex.DriverSchema` as the descriptor system:



To digress for a moment, the parameters here are interpreted as follows:

Application Name

This will serve as the XHTML `<title>` of the web application. By default, this string is also used to create the application client and responder class names.

Descriptor System

This must be a subclass of `ActiveRecordDescriptorSystem`. The class can also be created by the Glorp Mapping tool. For details on how to do this, refer to the *Glorp Developer's Guide*.

Package

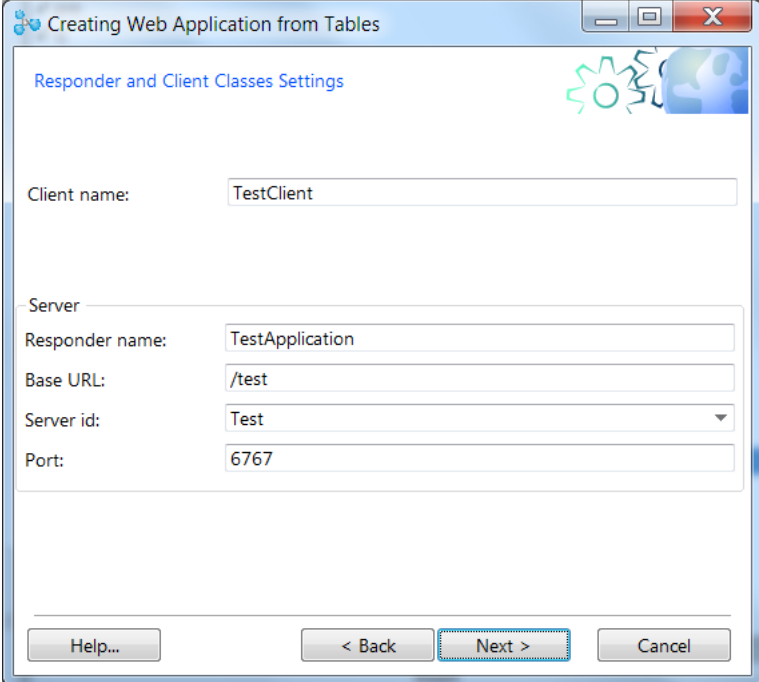
The package in which client and responder classes are placed. If the named package does not exist in the working image, it will be created.

Namespace

The name space in which new classes are placed.

When you are ready, click **Next**.

4. On the **Responder and Client Classes Settings** page, the default settings should suffice:



The screenshot shows a Windows-style dialog box titled "Creating Web Application from Tables". Inside, there's a tab labeled "Responder and Client Classes Settings". The form contains the following fields and values:

- Client name: TestClient
- Server (grouped header):
 - Responder name: TestApplication
 - Base URL: /test
 - Server id: Test (dropdown menu)
 - Port: 6767

At the bottom of the dialog are four buttons: "Help...", "< Back", "Next >" (which is highlighted with a blue border), and "Cancel".

These parameters may be interpreted as follows:

Client Name

The name for creating the subclass of ApplicationClient.

Responder name

The name for creating the subclass of AppeX.Application.

Base URL

The path is used as a base URL to all requests.

Server id

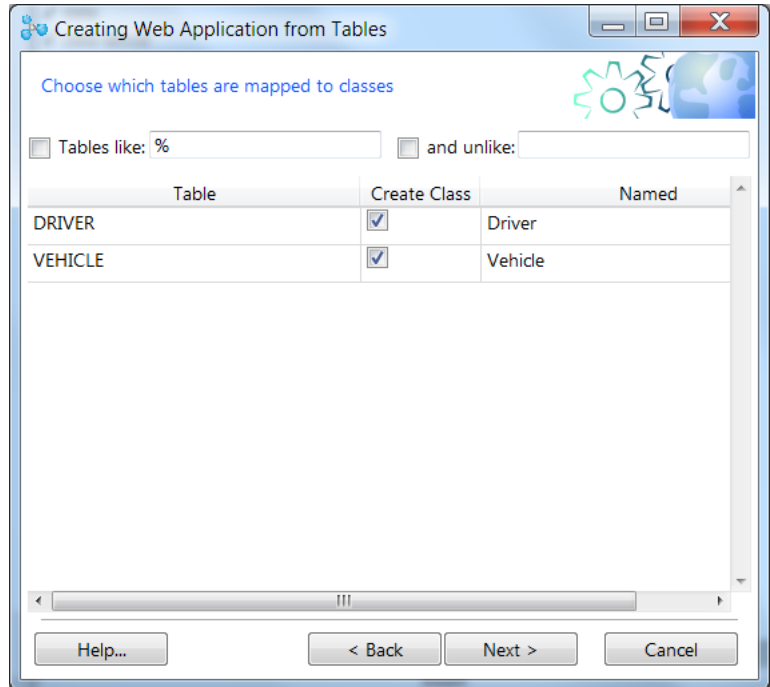
The name of a SiouX.Server, either pre-existing or new.

Port

The port on which the server listens.

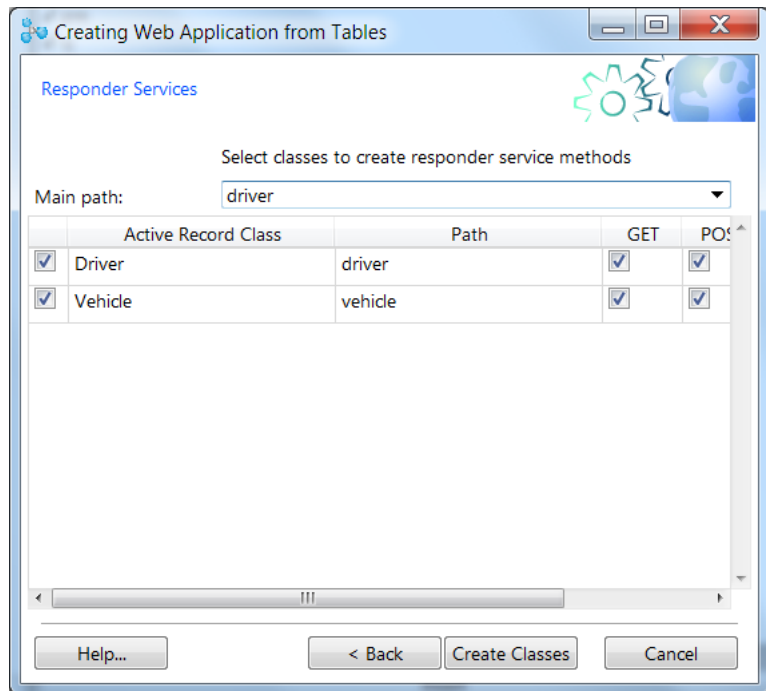
Click **Next**.

5. On the **Choose which Tables are Mapped to Classes** page, select the tables for which you would like the tool to create domain classes. For the present example, select the **DRIVER** and **VEHICLE** tables, as follows:



Click **Next**.

6. On the **Responder Services** page, select the application services corresponding to each table. for example:



These application services are represented by the following HTTP methods, which will be created by the tool:

GET

Retrieves all table records and displays them using a list view

POST

Updates a database record

PUT

Adds a new database record

DELETE

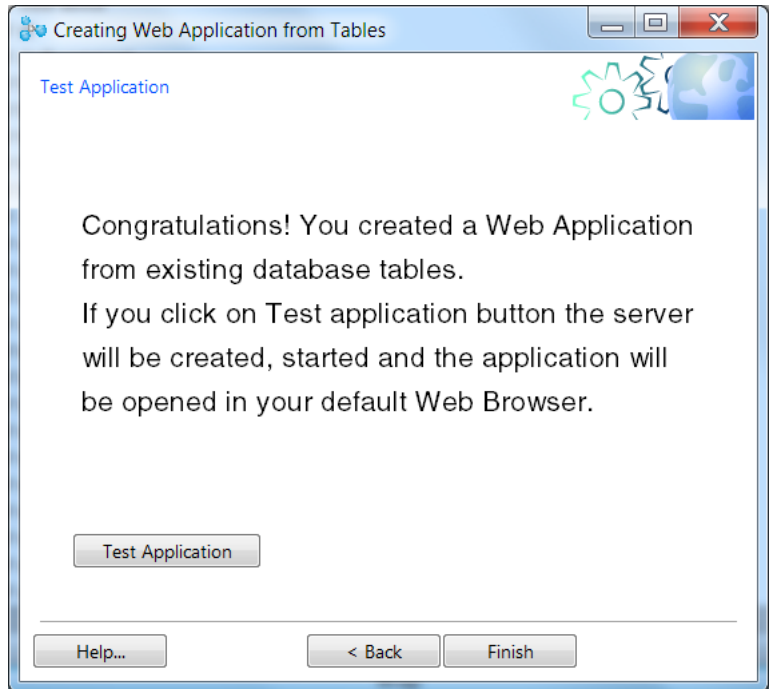
Deletes a database record

When the application is opened from a client's browser, the **Main path** is used to send the first web request.

When you have chosen all the services you wish to create, click the **Create Classes** button.

At this point, your web application is created. A more detailed explanation of the classes and methods created is provided below.

7. You can open the newly-created application on the **Test Application** page.



Clicking on the button uses `http://localhost:6767/test#list/driver` to open the default web browser on the newly-created responder.

The first page displays a list of drivers with vehicles. The **Vehicles** column uses the default Smalltalk `#printOn:` method for the instances of `Vehicle`.

8. To improve the display of **Vehicles**, open a System Browser and add the following method to class `Vehicle`:

```
printOn: aStream  
aStream  
  nextPutAll: make;  
  nextPutAll: ' ';  
  nextPutAll: model;
```

```
nextPutAll: ' mileage: ', mileage printString
```

Refresh the page in the web browser.

How it Works

In this example, clicking on the **Create Classes** button creates the following classes in the **Test** package:

Domain Classes from the Database Tables

In our case, the **Driver** and **Vehicle** classes.

TestApplication

The application class with service methods on the instance side, and configuration methods on the class side.

TestClient

The class that contains JavaScript code for the client side of the web application. This includes the functions: `buildHtml()`, `openMainView()`, and `mainPath()`.

The `buildHtml()` function is the main point at which you start building your page. You can edit this function, changing it to suit your application needs. By default, `buildHtml()` is defined as follows:

```
buildHtml() {
  //The method is the start point for the application.
  var scaffold, response;
  scaffold = new AppeX.Scaffold(this);
  //The first request returns the application database metadata.
  response = this.GET("initializeActiveRecordApplication");
  response
    .onSuccess(function (dataCollection) {
      //The information includes list of all database tables
      // and table attribute types.
      //The information is saved in the global this.objectTypeInfoos
      // and used by the framework to create labels and data entry fields
      // based on table column types
      scaffold.objectTypeInfoos = dataCollection;
      //Create and open the main application page
      scaffold.openMainView();
    })
    .onError(function(error){
```

```
//Set renderer to display the error
scaffold.setRenderer();
scaffold.renderer
.startElement("div")
.text( error.statusText + ":" + error.response)
.finishElement();
});
}
```

Framework

Using Scaffolding to build a web application primarily involves the judicious overriding of default framework methods, and on occasion the creation of specialized presenter classes. To understand the how and when of this, we need to explore some features of the Scaffolding framework in more detail.

Presenters

Presenter classes use Active Record schema information to render HTML pages. Presenter classes delegate the manipulation of XHTML elements to a `Renderer`. Common behavior is implemented by the abstract class `GenericPresenter`.

Specialized Presenters exist for listing, detailing, editing, and selecting domain objects. They handle the composition of the correct HTML for rendering on the client. They are “owned” by Scaffolding (which is itself owned by the `ApplicationClient`), and each presenter include a reference back to the `ApplicationClient`. Presenters collaborate with a `Renderer` (`JQueryRenderer`, by default) for creating JavaScript/HTML to display domain objects in the client's browser. In fact, the code generation is done inside the client's browser.

The Scaffolding framework dictates the names of the presenters, following the patterns of `List*`, `Detail*`, `Edit*`, and `Selection*`. The four concrete types of display presenters are thus: `ListPresenter`, `DetailPresenter`, `EditPresenter`, and `SelectionPresenter`. (`DetailPresenter` and `EditPresenter` are specializations of the more general `ObjectPresenter`.)

For each of these, a `JavascriptCode` class is implemented, which corresponds to the layout of a particular web page. That is, when

one of these Presenters is being used to render the page, only that Presenter is shown and none of the others.

By design, Presenters are highly configurable both through subclassing and overriding methods, but for the moment we shall only describe their basic functionality. To make this more concrete, we'll explore an example Genealogy application at the end of this section.

The Four Types of Presenters

ListPresenter

Displays a list of a objects of a particular Active Record type in tabular form, with one object per row and one attribute per column. Rows (domain objects) can be added (with an **Add New** button on the top of the table).

Domain objects can be inspected by clicking on the first column label for each object, rendered as a hyperlink, which opens up an `ObjectPresenter` for the linked object. The Genealogy example (discussed later) shows how rows can be removed (with **Delete** buttons on the end of each row).

DetailPresenter

`ObjectPresenters` come in two flavors: the `DetailPresenter` and the `EditPresenter`. The names of these Presenters indicate both their appearance on a web page and their function. Each of these presenters displays a single Active Record object in tabular mode, with the label of each attribute as the first column and a representation of the value as the second column.

The `DetailPresenter` allows only viewing, not modification of the object. If its attributes contain other domain objects such as values, those values in turn are presented as hyperlinks which can be opened to reach another `DetailPresenter`, theoretically ad-infinitum. The `DetailPresenter` also provides a **Close** button, to return the user to the previous page that was used to open the current view, and an **Edit** Button, that will open a `EditPresenter`, on the currently-displayed object.

EditPresenter

The `EditPresenter` allows for modification of the currently-viewed object. In the case of an attribute that contains a simple datatype (i.e. not a domain object or list), modification happens “in place”. To handle the case in which an attribute is not a simple datatype, there are several possibilities:

1. The attribute contains another domain object.

In this situation, the `EditPresenter` displays two buttons: **Create New...** and **Select Existing...**. In addition, if the attribute already contains a value, that value (domain object) can be opened, which invokes a new `EditPresenter` on it.

The **Create New...** button opens another `EditPresenter` on an “empty” object of the appropriate type, which can be accepted or cancelled (via **Accept** and **Cancel** buttons) once the required values are entered. The **Select Existing...** button opens a `SelectionPresenter` on a list of instances of the appropriate domain object type. Once the selection has been accepted (or alternatively, cancelled), the user is returned to the `EditPresenter` screen.

2. The attribute contains a list of other domain objects.

For the case in which the attribute represents a list of “owned” domain objects, then the **Create New...** button above is replaced by an **Add New...** button, which as before opens an `EditPresenter` on a new instance, but now when accepted the new instance is added to the list of owned objects rather than replacing the single owned object. Similarly, the **Select Existing...** button will in this situation add to the list of owned instances, rather than replacing it.

SelectionPresenter

The functionality of the `SelectionPresenter` has essentially been detailed in the discussion of the `EditPresenter`. The selection allows the user to choose one item from a list of the appropriate type of domain object, to be used as the value of a specified attribute. On the `SelectionPresenter` screen, the user is given the option to **OK** or **Cancel** their selection(s).

Naturally, the developer can set criteria for the objects that populate the list of possible choices.

Domain Objects

For an understanding of the Scaffolding framework, note that the phrase “domain object” refers somewhat interchangeably to:

1. The Active Record objects that describe the problem domain on the Smalltalk side (generally in a compositional-hierarchy pattern, that is, complex structures of domain objects that "own" other objects, such as a `Person` object might own a collection of `Cars`).
2. A corresponding domain object in the client's browser. These objects are given a type that corresponds to the class of the server instance.

For example, a `FamilyTree` object on the server might have a corresponding instance in the client, whose instantiation would invoke the code `new DomainObject()` with type information, and subsequently assign values to it.

Page Flow

The task of designing the flow of control between pages or screens is simplified as Scaffolding provides a simple way to structure CRUD applications. When a client is presented with a list of domain objects in the web browser, the framework also provides the mechanism to navigate through complex objects, inspecting or "drilling down" from a starting point to view the details of any particular object of interest.

For example, in a Genealogy application, the starting point might be a list of family trees. From there, navigation through the structure of associated domain objects is facilitated by Scaffolding, following a few simple patterns.

From the user's perspective, the opportunity to "drill down" into a domain object is displayed as a link on the current page. If the page is displaying a list of objects which are all of the same type (i.e. the page is a `ListPresenter`), where each item in the list is a specific

instance, then only the first column might display a link to inspect that particular object in more detail.

The value displayed in the cell of this first column is meant to be information of immediate interest about the instance represented by the entire row, e.g. its identity (though uniqueness is not enforced). At the implementation level, the content of each link defaults to the `#toString` of the corresponding domain object.

In a detailed view, the `DomainObject` instances that are stored as attributes of the domain object on the current screen (that was itself reached through either a previous "drill down" or by opening a **Create New...** screen), are drilled into by clicking on the display value of these attributes.

In summary, screens are displayed sequentially, with each one is reached either through drilling down with a link or pressing a **Create** or **Select** button.

Each successive screen is dedicated to one of the four following purposes (in no particular order of screen flow): (1) display a list of domain objects, all of which are of the same type; (2) view a single domain object in detail, and make available each of its complex attributes for "drill down" inspection; (3) edit a domain object, making its simple attributes available for change and its complex attributes available for "drill down" (still in edit mode) or for replacement (e.g. through "select existing" replacement); (4) select a domain object, so that it will replace the previous domain object (or blank) on the slot of the domain object featured in the previous screen.

Example: A Genealogy Application

Let's say you want to build a genealogy application that also provides both persistent storage and an in-browser CRUD experience for your users, so that they can manage information about their family trees. Here, "CRUD" implies the ability to create new family trees for inclusion in the database, to read them back, update them with new and/or corrected information, and delete them (though not necessarily in that order).

For this, you could of course create your own web pages using standard AppEx, starting by implementing your own `GenealogyClient` and writing your own custom page/screen flow logic using JavaScript. However, the Scaffolding framework offers a simple alternative to all of these development tasks.

Preliminaries

The following discussion presupposes that you've created, or know how to create, a web application using Active Record on the server, that already supports the GET, POST, PUT, and DELETE transactions. For details, see the discussion of [Active Record](#).

For the purposes of this example, we'll further presuppose that you've created the `GenealogyApplication` example, with its associated persistence methods, but that you haven't yet created a UI for this application. That is, you've already created the classes `GenealogyApplication`, `GenealogySchema`, `FamilyTree`, `LifeEvent`, etc., but haven't created any of the `*Client` or `*Presenter` (e.g., `GenealogyClient` or `FamilyTreeListPresenter`).

For reference, a complete, functional version of the Genealogy application is provided in the `AppEx-Examples-Scaffolding` package.

Creating Class `GenealogyClient`

To start building the client side presentation layer, we suggest [Using the Create Web Application Tool](#). Alternatively, you can create class `GenealogyClient` using the System Browser.

As an AppEx application class, `GenealogyApplication` names `GenealogyClient` as its `#applicationClientClass`. However, a Scaffolding application differs slightly in that this client is not used as the primary web page for the application. Instead, the Scaffolding approach is to dedicate a specific page to each "type of display" of an object.

For this, class `GenealogyClient` must delegate the display and editing of the domain objects (the client-side nomenclature for the "active record" objects on the server) to special subclasses of `JavaScriptCode`, by itself only implementing a few interesting, control-type methods. These methods include `buildHtml()` (the starting point for the application), `mainPath()` (naming the first type of object to

be displayed), `#shouldBuildBreadCrumbs` (for backtracking through the hierarchy of already displayed objects), and some overrides to the default labels for objects and their properties.

In class `GenealogyClient`, implement `buildHTML()` to create a client-side (JavaScript) instance of `AppEx.Scaffold`, which is basically a bundle of methods like `popView` and `setRenderer`. Don't worry if you don't yet understand exactly what each these framework methods is supposed to do at this point. The Scaffolding framework will perform its basic functionality, building the `viewStack` and delegating all the HTML construction to the default renderer, without you needing to implement anything extra.

Rather than working out how to compose a suitable `buildHTML()`, just cut-and-paste it from another scaffolding application, `DriverClient`. This method going to be the same for any similar CRUD application, unless it has been customized.

Overriding Special Methods

The next decision point is whether or not to build breadcrumbs. Let's do so by implementing `GenealogyClient>>shouldBuildBreadCrumbs()`, as follows:

```
shouldBuildBreadCrumbs() {  
  return true;  
}
```

We should also specify the page, or equivalently, the type of object, that the user sees when first opening the application. Since this is an application for displaying family trees and their contents, let's select a list of family trees by defining `GenealogyClient>>mainPath` as:

```
mainPath() {  
  return 'familytree';  
}
```

Creating Labels

You can override labels for any of the attributes on any of the objects in your domain (i.e. family trees, personages, dwellings, etc)

by implementing methods in class `GenealogyClient` that conform to a specific pattern recognized by the Scaffolding framework.

For example, let's consider the method: `labelFor_familytree_description`:

```
labelFor_familytree_description () {  
    return "Line";  
}
```

Note that there are no direct senders of this method. Instead, Scaffolding examines the class for methods starting with `labelFor_` and invokes them programmatically. The pieces of the method name identify the desired label substitution, specifying domain types and their attributes. (In this example, `familytree` names the domain object, and `_description` names an attribute). If methods named by this `labelFor_*` convention exist in `GenealogyClient`, Scaffolding invokes them whenever a label is needed for the attribute.

At this point, the application can be viewed in a web browser.

Working with Scaffolding

Working with Scaffolding entails customizing its basic functionality by creating subclasses and adding methods that follow a small number of specific patterns.

The subclasses you create are all inheritors of `GenericPresenter`, itself a subclass of `JavaScriptCode`. As such, the methods you add to specialize the behavior of your application are all client-side JavaScript functions.

To illustrate and explain these patterns, we shall continue to refer to the Genealogy application, which can be loaded in its entirety with the parcel `AppeX-Examples-Scaffolding`.

Creating Custom Presenter Classes

Using Scaffolding, building an application typically involves customizing pages/screens for listing, selecting, or editing objects by creating new Presenter classes. Each Presenter class provides a specific function for a specific type of domain object.

Once you have chosen the type of page that you are going to specialize — e.g., `ListPresenter`, `SelectionPresenter`, `EditPresenter`, or `DetailPresenter` —, create a subclass of that `Presenter`, using the class name of the domain object as the prefix of the new subclass. Next, implement an instance-side JavaScript method `#acceptedPaths`. This method specifies the type(s) of domain object the `Presenter` can manage.

To illustrate, let's consider the start page for the Genealogy application. For this, we create the class `FamilyTreeListPresenter` as a subclass of `ListPresenter`. Define an `acceptedPaths()` method that returns an array with `"familytree"`, e.g.:

```
acceptedPaths() {  
  return ["familytree"];  
}
```

With this subclass and method defined, Scaffolding can generate the client-side JavaScript whenever a list of family trees is to be presented.

To override the default title for the screen, define `getTitleDetails()` to return the string `"Genealogies"`, e.g.:

```
getTitleDetails() {  
  // Answer the details of the title. Normally, this is the path of the Active Record  
  // object(s)  
  // Subclasses may override.  
  return "Genealogies"  
}
```

And implement `getTitleAction()` to return the string `"Browse"`, e.g.:

```
getTitleAction() {  
  // Answer the title of this presenter. Subclasses are responsible.  
  return "Browse"  
}
```

With these methods defined, the web page should open with the title **"Browse Genealogies"**.

Labels for the columns can be overridden by implementing framework-invoked methods that follow the pattern `labelFor_*`, where

* is the name of the attribute that the column is meant to display. For example, you can implement `labelFor_rootPersonage` to return the string "Founder" in order to override the default column header for the `rootPersonage` attribute.

Implementing the Delete Function

Perhaps the most interesting method to implement in the `FamilyTreeListPresenter` is `deleteObject()`:

```
deleteObject(object) {
  var self = this,
      args = {};

  args.input = {};
  args.input.id = object.id;

  this.client
    .DELETE(this.path, args)
    .onSuccess(function (updatedObjects) {
      var index;
      $.debug("Delete " + self.path + " success");
      index = self.objects.indexOf(object);
      self.objects.splice(index, 1);
      self.setRenderer();
      self.buildView();
    });
}
```

If we implement this method, the `ListPresenter` builds an extra cell with a button to delete the object (a `FamilyTree`). Because deleting rows from the database can be tricky, depending on the context in which objects are viewed and the relational constraints between different tables, it is the responsibility of our subclass to implement an appropriate `deleteObject()` method.

In this case, we call on the server to delete the object on the database, then when the client is notified of success, we remove the family tree from the list of family trees and redraw the screen.

Customizing the Presentation of an Attribute

Scaffolding enables you to customize the presentation of any attribute of a domain object, by defining methods that follow the `buildCellDataFor_*` pattern. This works regardless of whether the attribute is a value, or contains another domain object.

To illustrate, let's look at class `PersonageEditPresenter`, and how it presents the `relation` attribute.

Class `PersonageEditPresenter` overrides `buildCellDataForRelation()` to add more delete buttons, which are actually "remove" buttons. That is, the corresponding "delete" methods are implemented to not actually delete domain objects from the database, but rather to remove the relation between the `Personage` instance and the object. E.g.:

```
deleteParent(attributeName, index, type) {
  // An example of a custom service
  var self = this,
      args = {},
      path;

  args.id = this.object.id;
  args.type = type;
  this.client
    .DELETE("removeParent", args)
    .onSuccess(function (statusString) {
      $.debug("Delete " + path + " success");
      self.deleteRelationFromObject(attributeName, index);
    });
}
```

The corresponding server-side code for `removeParent` must be implemented by the developer, i.e., `GenealogyApplication >> removeParentFrom:relation:`.

Other methods in `PersonageEditPresenter` that follow the `buildCellDataFor_*` pattern include `buildCellDataFor_gender()`, which specifies that the Presenter utilizes a radio button for input, and `buildCellDataFor_dayOfBirth()`, which specifies using a third-party Date widget for input by invoking the Scaffolding method `buildDateCellDataForAttributeNameAndValue()`.

Creating Domain Object Relationships

Scaffolding provides a means to create relationships between domain objects, via the `addExistingRelation_*` pattern. You can define methods which follow the pattern of `addExistingRelation_ + attributeName`, which invokes the framework method `selectExistingRelations()` to open a `SelectionPresenter` limited to appropriate choices.

For example, the method `addExistingRelation_mother()` opens a `SelectionPresenter` limited to the available selections for mother, female, and older than the current personage.

Validation with Custom EditPresenters

To specify and validate changes to the attributes of domain objects, you can subclass `EditPresenter` and implement an `acceptedPaths()` method that returns an Array of paths to a domain object. This method has the effect of making its subclass the preferred editor for the corresponding domain object(s).

For example, the class definition would be:

```
Smalltalk.AppeX defineClass: #AvocationEditPresenter
  superclass: #{AppeX.EditPresenter}
  ...
```

And the method would be:

```
acceptedPaths() {
  return ["avocation"];
}
```

In the Genealogy application, examples of this include:

`AvocationEditPresenter`, `DwellingEditPresenter`, `LifeEventEditPresenter`, and `PersonageEditPresenter`, which are all subclasses of `EditPresenter`.

Client-Side Validation in Scaffolding

Scaffolding supports two different types of client-side validation. You can validate an individual field or a whole page.

Individual field validation executes immediately upon entering a value in the field, while whole page validation inhibits **Accept**-ing a

screen until all validation has been passed. The validation of a whole page is implemented by building the page with the **Accept** button disabled (if any of the validations have failed), and only re-enabling **Accept** when all validation tests succeed.

Consider a typical scenario in which you wish to ensure that a user-entered number lies between a minimum and a maximum value. The general approach to this requirement would be to place a constraint on the number entered (not allowing anything outside of the values). Alternately, you could implement a validation warning for any value entered outside of those values. This latter approach — using warnings — is provided by Scaffolding.

Validating Individual Fields: Five Options

Class `EditPresenter` supports five predefined types of immediate input validation. At runtime, these validations modify the page with text that describes the nature of the input validation failure, as soon as the input value is changed.

For the implementation details, examine the `validateInput()` method in class `EditPresenter`.

The first three checks are basic validations, followed by a check for a possible custom validation, and lastly validation for blank inputs:

1. **`validateType_Number(input, min, max)`**

Validates that the input is a number, and that it lies between `min` and `max` (optional values).

2. **`validateType_String(input, minLength, maxLength)`**

Validates that the string length lies between `min` and `max`.

3. **`validateType_Time(input, min, max)`**

Validates that Time must be in the format `'xx:xx:xx'` where the values are numbers, the hour is between 0 and 24, etc.

4. **`this.checkForCustomValidation`**

Asks the framework to check for a developer-implemented method of the form `validate_ + attributeName`; if present, it is executed.

5. **`this.checkIsNullableValidation`**

Checks whether a domain object's attribute is allowed to be blank by examining the type information for the object passed from the server to the domain.

Validating a Whole Page

The validation of a whole page is implemented by overriding the `validationOnAccept()` method in class `EditPresenter`. That is, by implementing this method in your own custom subclasses.

For example, the `PersonageEditPresenter` requires that any `Personage` records at least one life event, by implementing the following method:

```
validationOnAccept () {
    var lifeEvents,
        validationTextStrings = [];

    lifeEvents = this.object.lifeEvents && this.object.lifeEvents.relations;
    if (!lifeEvents || lifeEvents.length < 1) {
        validationTextStrings.push("Minimum of one life event required.")
    }
    return validationTextStrings;
}
```

Similarly, in class `AvocationEditPresenter`, we find an example of specifying that the application does *not* use the basic (default) validation for the `years` attribute:

```
attributesNotRequiringBasicValidation() {
    return ['years'];
}
```

Implement this method when you want to perform some special validation. For example, in this case, the basic validation is just that the the input is a number. In place of this, we could substitute the validation of a range of numbers by implementing the following method:

```
validate_years(input) {
    return this.validateType_Number(input, 1, 10);
}
```

```
}
```

Here, note also that when the framework first builds the input field, it takes into account the developer-specified type of the simple attribute. You can only enter numbers in this field, and some browsers will also show up and down arrows as a visual cue that users can select to increment or decrement the input value.

Chapter

8

Server Monitor

Topics

- [Using the Server Monitor](#)

Appex includes the Server Monitor, a browser-based tool for management of Sioux servers running in a VisualWorks image. The Server Monitor is a counterpart to the Sioux Web Server configuration tool, which may be used locally or remotely, and is especially useful for managing headless images. It may also be used as an example of a moderately complex web application developed with Appex, using jQuery.

Using the Server Monitor

To start and access the Server Monitor, load the `Appex-ServerMonitor` parcel, and evaluate the following in a Workspace:

```
(Server id: 'ServerMonitor')  
start.ExternalWebBrowser open: 'http://localhost:8001'.
```

Naturally, if you want to access a running Server Monitor from another computer, browse that computer's IP address or its network name. Don't forget to include the port 8001.

If no servers other than the Server Monitor are configured in your image, the web page displays two buttons: **Create a Server** and **Add a Preconfigured Server**. These buttons correspond to the toolbar icons in the Sioux Web Server configuration tool which carry the same labels. The **Ping** and **Help** buttons are initially disabled.

If one or more servers are already configured, an 'accordion' widget is displayed below the buttons, in which each row show details for a single server.

Creating a New Server

To create a new, minimal server, click the **Create a Server** button. A dialog opens prompting for the new server name. Enter it, e.g., **MyServer**.

Once this is done, the **Servers** accordion displays an expanded row with the newly-added server in focus. Three buttons at the top, labeled **Stopped**, **Debugging** and **Logging** correspond to the server checkboxes in the GUI Web Servers configuration tool. A fourth button labeled **Log Contents** will display the contents of the server log, if it exists.

Below the buttons are some basic server statistics. For now, your new server has no listeners and no responders, so the statistics will show zero requests and responses.

For a server to start accepting requests, it must have at least one listener. Click on the **Add HTTP** button under the **Listeners** label. A dialog opens, prompting for the address (the default **0.0.0.0** means listen on

all interfaces), and the port number. Enter a port number, e.g., **9999**, and leave the **Reuse Address** check box selected. Click **OK**.

You can now start the server by either clicking the **Stopped** button under the server name, or the listener's **Paused** button (the "play" icon).

Next, you'll need to create a Responder. Under the **Responders** label, click the **Create...** button. A dialog presents a list of classes to select a responder from. Select a responder class of your choice, e.g.:

MyNamespace.MyApplication. Once you have selected a class, you can enter the path of the responder. The default is /. Enter a different path, e.g., **/my-responder**. It must begin with a /. Click **Submit**.

Your server is now configured to serve requests at a specific port, and has a single responder. If you click on the responder path, a new browser tab opens with the contents served by that responder.

Removing a Server

When a server is selected in the Server Monitor tool, you can remove it from the image by clicking **Remove this Server...** If the server is running, you'll be prompted for confirmation. When a server is removed, all its listeners and responders are destroyed and the server is stopped.

Adding a Preconfigured Server

The SiouX framework on which Appex is built provides a mechanism to programmatically configure servers with listeners and responders, avoiding tedious manual configuration. Programmatic configuration and the loading/saving of server configurations is described in detail in the [Web Server Developer's Guide](#).

Here, we focus on the details relevant to Appex web applications. Using the Server Monitor, a server can be added manually (see: [Creating a New Server](#)) or from a registry of preconfigured servers.

To make a configuration for the server registry:

1. Extend the `Sioux.Server` class with a class method that gives your server a name and adds a listener. The `#server:` pragma is used to

declare the method as a server configuration. The name of the method itself is not important, as long as it doesn't conflict with any of the existing class methods in `Sioux.Server` API.

```
Sioux.Server class >> #myConfiguration: server  
<server: 'My Server'>  
server listenOn: 9999 for: HttpConnection
```

2. Add a class method to your application class. Again, the name of the method itself is not important. The pragma `#server:path:` declares the method to be a responder configuration, e.g.:

```
MyNamespace.MyApplication class >> myServerConfiguration  
<server: 'My Server' path: '/my-responder'>
```

In the Server Monitor, click the **Add a Preconfigured Server** button. A dialog opens to present a list of server configurations you may use to add your server. You should see an entry with the name **My Server** and `Sioux.Server` as the server class. Select it. The server list accordion should now have **My Server** in focus, showing a listener on port **9999** and your `MyApplication` as one of the responders. You can create additional responder configurations by adding more responder configuration methods to any of the existing responder classes in the system.

3. Add another responder configuration to your application:

```
MyNamespace.MyApplication class >> anotherConfiguration  
<server: 'My Server' path: '/another-responder'>
```

Back in Server Monitor, make sure **My Server** is in focus. Under the **Responders** label, click **Add...** and select **Another Responder** from the list. You now have two responders for your application, each processing requests from a different path.

Accessing your Application from the Server Monitor

If you have configured a server with at least one listener and at least one responder, you can access your application right from the Server Monitor tool.

Make sure your server is running (i.e., the first button under the name of your server shows **Running**, rather than **Stopped**).

Click on the link corresponding to your application under the **Responders** label. The browser will open a new tab with your application loaded.

Setting the Responders' Order

When a SiouX server receives an HTTP request, it queries a list of its responders, and the first one that answers true to the `#acceptRequest:` message is used to process the request. The default implementation (`Sioux.Request >> #acceptRequest:`) uses the request's path to determine the return value based on the path under which the responder is registered with the server. For example, the request path `/my-path` will be considered as a valid processing path for any of the following responder paths: `/mypath/pathA`, `/my-path/pathB`, and so on. As a result, if you have a number of responders registered with a server, you need to consider their paths to ensure correct dispatching of requests to responders. Generally, you will want to have the deeper paths listed before the shallower ones. That means that if you have a responder that uses `/` as a path, it must be last on the lists.

In the Server Monitor tool, you can reorder the responders by dragging. When you move the mouse cursor over the responders list, the cursor will change to a 'drag' symbol. Drag a responder and move it to a different position on the list. When you release the mouse button, an **Update** button will appear above the responder list. Click it to update the responder order on the server.

You can also determine the order in the configuration methods by including an optional `#position:` pragma. The argument to the `#position:` pragma is a number that determines the relative order in which a given instance of the responder will be added to a preconfigured server. Lower numbers will be listed before higher numbers. When the `#position:` arguments for two different responders are equal, the order will be undetermined. If no `#position:` pragma is present, the value of zero is used.

To illustrate, add a new responder configuration method that sets the relative position to a high number, and set the responder path

to /. Effectively, the responder added with this configuration method will become the 'default' or 'catch all' responder for your server:

MyNamespace.MyApplication class >> #defaultConfiguration

"The position: argument has to be the highest for any of the responders configured for 'MyServer'. Any responder with position larger than 99999 will be ignored because the root path '/' will act as a catch-all for all HTTP requests."

```
<position: 99999>
```

```
<server: 'My Server' path: '/'>
```


Index

"Hello World!" tutorial [5](#)

A

- Active Record
 - defined [63](#)
- AJAX [50](#)
- application
 - class [4](#), [10](#)
 - client class [4](#)
 - model [10](#)
 - services [18](#)
- application caching
 - explained [24](#)
- application services
 - application caching [23](#)
 - JavaScript [21](#)
 - JSON [21](#)
 - server-sent events [23](#)
 - static file [22](#)
 - types [18](#)
- architecture
 - of AppEx [4](#)

C

- class
 - AppEx.Application [11](#)
 - AppEx.ApplicationClient [12](#)
 - GenericJavascript [14](#)
 - JavascriptCode [13](#)
 - JavascriptObject [14](#)
- client-side behavior
 - implementing [10](#), [13](#)
- client-side request template
 - generating [37](#)
- conventions
 - typographic [vii](#)

E

- EventStream
 - API [54](#)

F

- fonts [vii](#)

H

- HTML documents
 - components [15](#)
 - generating [15](#)
 - templating [17](#)

J

- JavaScript
 - organization [33](#)
 - using third-party libraries [36](#)
 - writing [32](#)

L

- library
 - CoreCode [15](#)

M

- messages
 - sending synchronously [51](#)
 - sending synchronously vs. asynchronously [50](#)
 - sending to a server [50](#)
 - setting up event handlers [50](#)

N

- notational conventions [vii](#)

P

parcels

- AppeX [4](#)
- loading [5](#)
- optional [5](#)

R

requests

- dispatching [26](#)

requests and responses

- accessing [24](#)

S

Scaffolding

- defined [73](#)

Server Monitor [104](#)

Server-Sent Events (SSE)

- AppeX implementation [52](#)
- registering interest [52](#)
- with session management [54](#)
- without session management [53](#)

service methods

- declaring [18](#)

service types

- adding custom [27](#)
- defining custom [25](#)
- predefined [19](#)

session keys

- defining [59](#)

session management

- accessing state [60](#)
- enabling [58](#)
- linking client and server [59](#)
- overview [58](#)
- setting expiration time [60](#)

special symbols [vii](#)

symbols used in documentation [vii](#)

T

tools support

- enabling [4](#)

typographic conventions [vii](#)

X

XmlHttpRequest

- using [50](#)